Publication Series of the John von Neumann Institute for Computing (NIC)
NIC Series Volume 33

John von Neumann Institute for Computing (NIC)

Parallel Computing: Current and Future Issues of High End Computing

Edited by

G. R. Joubert
Clausthal
Germany

W. E. Nagel
Dresden
Germany

F. J. Peters
Eindhoven
Netherlands

O. Plata
Malaga
Spain

P. Tirado
Madrid
Spain

E. Zapata
Malaga
Spain

NIC Series Volume 33

ISBN 3-00-017352-8

Die Deutsche Bibliothek – CIP-Cataloguing-in-Publication-Data
A catalogue record for this publication is available from Die Deutsche Bibliothek

Publisher: NIC-Directors
Distributor: NIC-Secretariat
Research Centre Jülich
52425 Jülich
Germany
Internet: www.fz-juelich.de/nic
Printer: Graphische Betriebe, Forschungszentrum Jülich

NIC Series Volume 33 ISBN 3-00-017352-8

Preface

The development of parallel computing technologies saw rapid changes in recent years. The development and employment of specialised hardware to achieve high performance was replaced by an increasing use of standard off-the-shelf processors that were interconnected by various network architectures to form parallel systems. Due to the cost advantages offered by standard components the price/performance of such systems made specialised hardware based systems uncompetitive.

Today we see a deepening of this trend. Standard processors increasingly utilise parallel technologies to achieve higher performance. On the one hand this trend can be seen in standard processors used for personal computers. These already incorporate various parallel computing technologies ranging from the functional design of processors themselves to multi-processor architectures. On the other hand one also sees a trend towards the development of specialised parallel processors for mass markets. The most noted example is the Cell processor designed for use in the next generation Sony Playstation. Through its high potential performance, its mass market and resulting low price this processor can impact the professional high speed computing market. First steps in this direction were taken already with the introduction of the Dual Cell-Based Blade by Mercury Computer Systems. A system based on four Cell processors was recently announced. This is only one example of a number of similar developments. To-day Intel, AMD, Microsoft, etc. are all moving in the direction of employing parallel technologies to an increasing degree. Intel has been doing this together with Hewlett-Packard for a number of years already. These activities are gaining in momentum.

The ***ParCo2005*** conference took place against the background of this fast changing scenario. Since the first ***ParCo*** conference in 1983 these events greatly influenced the development and application of parallel computing technologies, methods and tools in its widest sense.

In addition to the contributed papers a total of five mini-symposia on special topics and an industrial session formed part of the program.

As with all previous conferences in this series the emphasis of ***ParCo2005*** was again on quality rather than quantity. Thus all contributions were reviewed prior to and again during the conference. This approach has the disadvantage that proceedings only come available many months after the event. A further disadvantage is that authors who presented a paper may be confronted with the situation that their paper as presented may not be approved for publication in the proceedings. The current book thus does not contain all papers presented at the conference. Organisers of mini-symposia were given the choice to review and publish papers presented at the conference in the proceedings. In total three invited papers, 83 contributed papers and 24 papers from mini-symposia are included in this book.

The Editors are greatly indebted to the members of the International Program Committee as well as the organisers of the mini-symposia for their support in selecting and reviewing the large number of papers.

The organisers of the conference are also greatly indebted to the members of the Steering, Organising, Finance and Exhibition Committees for the time they spent in making this conference such a successful event.

Special thanks are due to the staff of the Department of Computer Architecture of the University of Malaga for their enthusiastic support. In this regard particular mention should be made of the key roles played by Rafael Asenjo and Luis F. Romero (University of Malaga) as well as Manuel Prieto (University Complutense, Madrid) in making this event such a great success.

Gerhard Joubert
Germany/Netherlands

Wolfgang Nagel
Germany

Frans Peters
Netherlands

Oscar Plata
Spain

Paco Tirado
Spain

Emilio Zapata
Spain

December 2005

CONFERENCE COMMITTEE

Gerhard R. Joubert (Germany/Netherlands) (Conference Chair)
Wolfgang E. Nagel (Germany)
Frans J. Peters (Netherlands)
Oscar Plata (Spain)
Francisco Tirado (Spain)
Emilio L. Zapata (Spain)

STEERING COMMITTEE

Frans J. Peters (Netherlands) (Chair)
Arndt Bode (Germany)
Friedel Hoßfeld (Germany)
Daniel A. Reed (USA)
Denis Trystram (France)
Mateo Valero (Spain)
Marco Vanneschi (Italy)
Hans Zima (Austria/USA)

ORGANISING COMMITTEE

Emilio L. Zapata (Spain) (Chair)
Oscar Plata (Spain) (Chair)
Rafael Asenjo (Univ. of Malaga, Spain)
Manuel Prieto, Univ. Complutense Madrid, Spain)
Luis F. Romero (University of Malaga, Spain)

FINANCE COMMITTEE

Frans J. Peters (Netherlands) (Chair)

SPONSORS

Hewlett Packard, S.L.
Silicon Graphics, S.A.
Bull, Spain, S.A.
University of Málaga
Spanish Ministry of Education and Science
Fundación General of University of Málaga
Málaga City Council

EXHIBITORS / PARTICIPANTS IN THE INDUSTRIAL TRACK

Hewlett Packard, S.L.
Silicon Graphics, S.A.
Bull, Spain, S.A.
Sun Microsystems Ibérica, S.A.
McGraw-Hill Interamericana de España, S.A.
Thomson Paraninfo, S.A.
Pearson, Prentice-Hall

PROGRAM COMMITTEE

Francisco Tirado (Spain) (Chair)
Wolfgang E. Nagel (Germany) (Chair)

Norbert Attig (Germany)
Henri Bal (Netherlands)
Dirk Bartz (Germany)
Achim Basermann (Germany)
Christian Bischof (Germany)
Petter E. Bjørstad (Norway)
Arndt Bode (Germany)
Mats Brorsson (Sweden)
José M. Carazo (Spain)
Barbara Chapman (USA)
Andrea Clematis (Italy)
Michel Cosnard (France)
Pasqua D'Ambra (Italy)
Luisa D'Amore (Italy)
Koen De Bosschere (Belgium)
Luiz DeRose (USA)
Andreas Deutsch (Germany)
Erik H. D'Hollander (Belgium)
Beniamino Di Martino (Italy)
Ramón Doallo (Spain)
Rüdiger Esser (Germany)
Thomas Fahringer (Austria)
Afonso Ferreira (France)
Salvatore Filippone (Italy)
Efstratios Gallopoulos (Greece)
Inmaculada García (Spain)
Michael Gerndt (Germany)
Lucio Grandinetti (Italy)
Volker Gülzow (Germany)
Rolf Hempel (Germany)
Vicente Hernández (Spain)
Lennart Johnsson (USA)
Odej Kao (Germany)
Christoph Kessler (Sweden)
Dieter Kranzlmüller (Austria)
Norbert Kroll (Germany)
Herbert Kuchen (Germany)
Alexey Lastovetsky (Ireland)
Pierre Leca (France)
Thomas Lippert (Germany)
Thomas Ludwig (Germany)
Emilio Luque (Spain)
Allen D. Malony (USA)
Tomás Margalef (Spain)
Djordje Maric (Switzerland)
Federico Massaioli (Italy)
Bernd Mohr (Germany)
Matthias Müller (Germany)
Almerico Murli (Italy)
Per Öster (Sweden)
Jean-Louis Pazat (France)
Wilfried Philips (Belgium)
Rolf Rabenseifner (Germany)
Thomas Rauber (Germany)
Michael Resch (Germany)
Yves Robert (France)
Dirk Roose (Belgium)
Gudula Rünger (Germany)
Masaaki Shimasaki (Japan)
Horst D. Simon (USA)
Henk Sips (Netherlands)
Tor Sørevik (Norway)
Craig Stewart (USA)
Erich Strohmaier (USA)
Domenico Talia (Italy)
Juan Touriño (Spain)
Jeffrey Vetter (USA)
Heinrich Voss (Germany)
Helmut Weberpals (Germany)
Philip Wilsey (USA)
Roland Wismüller (Germany)
Gabriel Wittum (Germany)
Albert Y. Zomaya (Australia)

CONTENTS

Invited Papers **1**

Supporting Large Scale Medical and Scientific Datasets 3
U. Catalyurek, S. Hastings, K. Huang, V.S. Kumar, T. Kurc, S. Langella, S. Narayanan, S. Oster, T. Pan, B. Rutt, X. Zhang, J.H. Saltz

Periscope: Advanced Techniques for Performance Analysis 15
M. Gerndt, K. Fürlinger, E. Kereku

The Mitosis Speculative Multithreaded Architecture 27
C. Madriles, C. García Quiñones, J. Sánchez, P. Marcuello , A. González

Grid Computing **39**

A Paradigm for Allocating Parallel Application Tasks to Heterogeneous Computing Resources on the Grid 41
B. Arafeh, K. Day, A. Touzene

Science Experimental Grid Laboratory (SEGL) Dynamical Parameter Study in Distributed Systems 49
N. Currle-Linde, U. Küster, M. Resch, B. Risio

On Scheduling in UNICORE - Extending the Web Services Agreement based Resource Management Framework 57
A. Streit, O. Wäldrich, P. Wieder, W. Ziegler

A Framework for Dynamic Adaptation of Parallel Components 65
J. Buisson, F. André, J.-L. Pazat

Towards a Distributed Scalable Data Service for the Grid 73
M. Aldinucci, M. Danelutto, G. Giaccherini, M. Torquati, M.Vanneschi

eNANOS: Coordinated Scheduling in Grid Environments 81
I. Rodero, F. Guim, J. Corbalán, J. Labarta

Parallel Program/Component Adaptivity Management 89
M. Aldinucci, F. André, J. Buisson, S. Campa, M. Coppola, M. Danelutto, C. Zoccolo

Load Balancing Support for Grid-enabled Applications 97
S. Rips

NewsGrid: Infrastructure and Services 105
S. Geisler, G.R. Joubert

Provision of Fault Tolerance with Grid-enabled and SLA-aware Resource Management Systems 113
F. Heine, M. Hovestadt, O. Kao, A. Keller

Air Pollution Forecast on the HUNGRID Infrastructure 121
R. Lovas, J. Patvarczki, P. Kacsuk, I. Lagzi, T. Turányi, L. Kullmann, L. Haszpra, R. Mészáros, A. Horányi, A. Bencsura, Gy. Lendvay

Distributed Shared Memory in a Grid Environment 129
J.P. Ryan, B.A. Coghlan

Hierarchical and Reliable Multicast Communication for Grid Systems 137
N. Ranaldo, G. Tretola, E. Zimeo

Building Interoperable Grid-aware ASSIST Applications via Web Services 145
M. Aldinucci, M. Danelutto, A. Paternesi, R. Ravazzolo, M. Vanneschi

Performance Evaluation & Analysis 153

Performance Analysis of Parallel Applications with KappaPI 2 155
J. Jorba, T. Margalef, E. Luque

A Comparative Evaluation of Two Techniques for Predicting the Performance of Dynamic Enterprise Systems 163
D.A. Bacigalupo, S.A. Jarvis, L. He, D.P. Spooner, D. Pelych, G.R. Nudd

Scheduling for Heterogeneous Networks of Computers with Persistent Fluctuation of Load 171
R. Higgins, A. Lastovetsky

Analysis and Optimization of Yee_Bench Using Hardware Performance Counters 179
U. Andersson, P. Mucci

Holistic Hardware Counter Performance Analysis of Parallel Programs 187
B.J.N. Wylie, B. Mohr, F. Wolf

Ring Algorithms on Heterogeneous Windows-based Clusters with Various Message Passing Environments 195
A. Clematis, A. Corana

Phase-Based Parallel Performance Profiling 203
A.D. Malony, S.S. Shende, A. Morris

Tracing the Cache Behavior of Data Structures in Fortran Applications 211
L. Barabas, R. Müller-Pfefferkorn, W.E. Nagel, R. Neumann

Algorithms 219

A Parallel Variant of the Gram-Schmidt Process with Reorthogonalization 221
V. Hernández, J.E. Román, A. Tomás

Auto-Optimization of Linear Algebra Parallel Routines: The Cholesky Factorization 229
L.-P. García, J. Cuenca, D. Giménez

A Parallel Algebraic Multigrid Preconditioner Using Algebraic Multicolor Ordering for Magnetic Finite Element Analyses 237
T. Mifune, N. Obata, T. Iwashita, M. Shimasaki

Parallel Newton-Type Iterative Methods Based on ILU Factorizations 245
J. Arnal, H. Migallón, V. Migallón, J. Penadés

Parallel Algorithm for Nonlinearly Unconstrained Optimization Based in Parametric Trees 253
I. Pardines, D.E. Singh, F.F. Rivera

Parallel Global Optimisation for Oil Reservoir Modelling 261
S. Gómez, N. del Castillo

Parallelization of an Algorithm for Finding Facility Locations for an Entering Firm Under Delivered Pricing 269
J.L. Redondo, I. García, P.M. Ortigosa, B. Pelegrín, P. Fernández

Concurrent Parallel Shortest Path Computation 277
D. Nussbaum, J.-R. Sack, H. Ye

OpenMP Parallelizations of Viswanathan and Bagchi`s Algorithm for the Two Dimensional Cutting Stock Problem 285
G. Miranda Valladares, C. León Hernández

A Parallel Adaptive Algorithm to Improve Precision of Time Series Identification 293
J.A. Gómez, M.A. Vega, J.M. Sánchez, J.M. Granado

Towards Robustness in Parallel SAT Solving 301
W. Blochinger

Asynchronous Iterative Computations with Web Information Retrieval Structures: The PageRank Case 309
G. Kollias, E. Gallopoulos, D. Szyld

Solving Real Life Applications With High Accuracy 317
C.A. Hölbig, P.S. Morandi Jr., D.M. Claudio, T.A. Diverio

Improving Ease of Use in BLACS and PBLAS with Python 325
T. Drummond, V. Galiano, V. Migallón, J. Penadés

Parallelization of GSL on Clusters of Symmetric Multiprocessors 333
J. Aliaga, F. Almeida, J.M. Badía, S. Barrachina, V. Blanco, M. Castillo, R. Mayo, E.S. Quintana, G. Quintana, C. Rodríguez, F. de Sande, A. Santos

Simulations **341**

Parallel Program Complex for High Reynolds Unsteady Flow Simulation 343
B.N. Chetverushkin, E.V. Shilnikov

Data Structures and Mesh Processing in Parallel CFD Project GIMM 351
B.N. Chetverushkin, V.A. Gasilov, S.V. Polyakov, M.V. Iakobovski, E.L. Kartasheva, A.S. Boldarev, A.S. Minkin

Radiative Gas Dynamics Parallel Computing Using Unstructured Meshes 359
B.N. Chetverushkin, V.A. Gasilov, O.G. Olkhovskaya, S.V. D'yachenko, E.L. Kartashova, A.S. Boldarev, V.V. Valko

Performance Analysis and Visualization of the N-Body Tree Code PEPC on Massively Parallel Computers 367
P. Gibbon, W. Frings, S. Dominiczak, B. Mohr

Experiments with a Parallel Monte Carlo Simulation of Space Cosmic Particles Detector 375
F. Almeida, F. de Sande, C. Delgado, R. García-López

Parallel Simulation of Tsunamis Using a Hybrid Software Approach 383
X. Cai, G.K. Pedersen, H.P. Langtangen, S. Glimsdal

Parallel Simulations of Underground Flow in Porous and Fractured Media 391
A. Beaudoin, J.R. De Dreuzy, J. Erhel, H. Mustapha

A Parallel Software for a Saltwater Intrusion Problem 399
E. Canot, C. de Dieuleveult, J. Erhel

A High-Performance Parallel Device Simulator for High Electron Mobility Transistors 407
N. Seoane, A. J. García-Loureiro, K. Kalna, A. Asenov

Real-Time Simulation for Laser-Tissue Interaction Model 415
L.F. Romero, O. Trelles, M.A. Trelles

Computer Simulation of the Acoustic Impedance of Modern Orchestral Horns 423
A. Benoit, J.P. Chick

Parallelization of the C++ Navier-Stokes Solver DROPS with OpenMP 431
C. Terboven, A. Spiegel, D. an Mey, S. Gross, V. Reichelt

Optimization of an Octree-based 3-D Parallel Meshing Algorithm for the Simulation of Small-Feature Semiconductor Devices 439
J.J. Pombo, M. Aldegunde, A.J. García-Loureiro

Large Scale Simulation of Ideal Quantum Computers on SMP-Clusters 447
G. Arnold, T. Lippert, N. Pomplun, M. Richter

Cluster Computing **455**

Adaptive Selection of Communication Methods to Optimize Collective MPI Operations 457
O. Hartmann, M. Kühnemann, T. Rauber, G. Rünger

Fault Tolerant Master-Worker over a Multi-Cluster Architecture 465
J. Rodrigues de Souza, E. Argollo, A. Duarte, D. Rexachs, E. Luque

A Distributed Scheme for Fault-Tolerance in Large Clusters of Workstations 473
A. Duarte, D. Rexachs, E. Luque

An Automated Approach to Improve Communication-Computation Overlap in Clusters 481
L. Fishgold, A. Danalis, L. Pollock, M. Swany

I/O & Databases **489**

QoS-aware Query Processing in Cluster-based Image Databases 491
U. Rerrer, O. Kao

A New Algorithm for Join Processing with the Internet Transfer Delays 499
K. Imasaki, S. Dandamudi

Comparing Two Parallel File Systems: PVFS and FSDDS 507
J. Buenabad-Chávez, S. Domínguez-Domínguez

pCFS: A Parallel Cluster File System 515
P.A. Lopes, P.D. Medeiros

Parallel I/O Optimization for an Air Pollution Model 523
D.E. Singh, F. García, J. Carretero

A Parallel Data Management Layer for Data Mining 531
M. Coppola, P. Pesciullesi, L. Presti, R. Ravazzolo, M. Vanneschi

Compiler Techniques **539**

Reducing Cache Misses by Loop Reordering 541
E. Herruzo, G. Bandera, E.L. Zapata, O. Plata

Performance Evaluation of Barrier Techniques for Distributed Tracing Garbage Collectors 549
J.M. Velasco, D. Atienza, K. Olcoz, F. Catthoor

A New Strategy for Shape Analysis Based on Coexistent Links Sets 557
A. Tineo, F. Corbera, A. Navarro, R. Asenjo, E.L. Zapata

Optimal Tile Size Selection Guided by Analytical Models 565
B.B. Fraguela, M.G. Carmueja, D. Andrade

Pack Transposition: Enhancing Superword Level Parallelism Exploitation 573
C. Tenllado, L. Piñuel, M. Prieto, F. Catthoor

Applications **581**

High Volume Colour Image Processing with Massively Parallel Embedded Processors 583
J. Jacobs, W. Bond, R. Pouls, G.J.M. Smit

Parallel Endmember Extraction Techniques Applied to a Self-Organizing Neural Network for Hyperspectral Image Classification 591
D. Valencia, A. Plaza, R.M. Pérez, M.C. Cantero, P. Martínez, J. Plaza

JPEG2000 Optimization in General Purpose Microprocessors 599
C. García, C. Tenllado, L. Piñuel, M. Prieto

A Parallel IMAGE Processing Server for Distributed Applications 607
A. Clematis,D. D`Agostino, A. Galizia

A Speculative Parallel Algorithm for Self-Organizing Maps 615
C. García, M. Prieto, A. Pascual-Montano

Genomic-Scale Analysis of DNA Words of Arbitrary Length by Parallel Computation 623
X.Y. Yang, A. Ripoll, V. Arnau, I. Marín, E. Luque

Parallel Implementation of SEMPHY-a Structural EM Algorithm for Phylogenetic Reconstruction 631
E. Li, Z. Ouyang, X. Deng, Y. Zhang, W. Chen

Exploiting Parallelism on Irregular Applications Using the GPU 639
M. Ujaldón, J.H. Saltz

Biomedical and Civil Engineering Experiences Using Grid Computing Technologies 647
J.M. Alonso, V. Hernández, G. Moltó

Parallel Compression of 3D Meshes for Efficient Distributed Visualization 655
A. Clematis, D. D`Agostino, V. Gianuzzi

A Parallel Implementation of a 3D Reconstruction Algorithm for Real-Time Vision 663
J. Falcou, J. Sérot, T. Chateau, F. Jurie

Languages **671**

Using eSkel to Implement the Multiple Baseline Stereo Application 673
A. Benoit, M. Cole, S. Gilmore, J. Hillston

A Java/Jini Framework Supporting Stream Parallel Computations 681
M. Danelutto, P. Dazzi

Architecture **689**

Massively Parallel MIMD Architecture Achieves High Performance in a Spam Filter 691
O.R. Birkeland, M. Nedland, O. Snøve Jr.

Implementing Critical Sections with the Shared Explicit Cache System in the Shared Memory Parallel Architectures 699
T. Madajczak, H. Krawczyk

Scheduling **707**

A New Genetic Convex Clustering Algorithm for Parallel Time Minimization with Large Communication Delays 709
J.E.P. Sanchez, D. Trystram

Scheduling issues on IBM p690: Performance Analysis with the PARbench Environment 717
H. Dietze, W.E. Nagel, B. Trenkler

Minisymposium - Bioinformatics **725**

Implementation of Anisotropic Nonlinear Diffusion for Filtering 3D Images in Structural Biology on SMP Clusters 727
S. Tabik, E. M. Garzón, I. García, J.J. Fernández

Services Integration and Task-scheduling in Bioinformatics Grids 735
S. Ramírez , E. de-Andrés , I. Navas-Delgado , A.J. Pérez , J. Aldana , O. Trelles

Minisymposium – Network-on-Chip **743**

Networks on Chips: A Synthesis Perspective 745
F. Angiolini, P. Meloni, D. Bertozzi, L. Benini, S. Carta, L. Raffo

NoC Emulation on FPGA: HW/SW Synergy for NoC Features Exploration 753
N. Genko, D. Atienza, G. De Micheli

Distributed Congestion Control for Packet Switched Networks on Chip 761
T. Marescaux, A. Rångevall, V. Nollet, A. Bartic, H. Corporaal

Versatile FPGA-Based Functional Validation Framework for Networks-on-Chip Interconnections Designs 769
J.B. Perez Ramas, D. Atienza, M. Peón, I. Magán, J.M. Mendías, R. Hermida

A New Model for NoC-based Distributed Heterogeneous System Design 777
F. Rincón, F. Moya, J. Barba, D. Villa, F.J. Villanueva, J.C. López

Minisymposium - Skeletons **785**

Integrating MPI-Skeletons with Web Services for Grid Programming 787
J. Dünnweber, A. Benoit, M. Cole, S. Gorlatch

Scalable Farms 795
M. Poldner, H. Kuchen

“Second-generation” Skeleton Systems 803
M. Danelutto

Domain Decomposition and Skeleton Programming with OCamlP3l 811
F. Clément, V. Martin, A. Vodicka, R. Di Cosmo, P. Weis

Mondriaan sparse matrix partitioning for attacking cryptosystems by a parallel block Lanczos algorithm — a case study 819
R.H. Bisseling, I. Flesch

Efficient representation and parallel computation of string-substring longest common subsequences 827
A. Tiskin

Skeletons for Recursively Unfolding Process Topologies 835
J. Berthold, R. Loogen

Towards Improving Skeletons in Eden 843
M. Hidalgo-Herrero, Y. Ortega-Mallén, F. Rubio

Reasoning About Skeletons in Eden 851
R. Peña, C.M. Segura

Merging Compositions of Array Skeletons in SAC 859
C. Grelck, S.-B. Scholz

Minisymposium - Tools **867**

Scalability of Visualization and Tracing Tools 869
J. Labarta, J. Giménez, E. Martínez, P. González, H. Servat, G. Llort, X. Aguilar

Performance comparison and optimization: Case studies using BenchIT 877
R. Schöne, G. Juckeland, W.E. Nagel, S. Pflüger, R. Wloch

Performance Analysis of One-sided Communication Mechanisms 885
B. Mohr, A. Kühnal, M.-A. Hermanns, F. Wolf

Runtime Checking of MPI Applications with MARMOT 893
B. Krammer, M.S. Müller, M.M. Resch

Automated Correctness Analysis of MPI Programs with Intel Message Checker 901
V. Samofalov, V. Krukov, B. Kuhn, S. Zheltov, A. Konovalov, J. DeSouza

The MPI/SX Collectives Verification Library 909
J.L. Träff, J. Worringen

Capturing Petascale Application Characteristics with the Sequoia Toolkit 917
J.S. Vetter, N. Bhatia, E.M. Grobelny, P.C. Roth

Author Index **925**

Invited Papers

Supporting Large Scale Medical and Scientific Datasets*

Umit Catalyurek[a], Shannon Hastings[a], Kun Huang[a], Vijay S. Kumar[a], Tahsin Kurc[a], Stephen Langella[a], Sivaramakrishnan Narayanan[a], Scott Oster[a], Tony Pan[a], Benjamin Rutt[a], Xi Zhang[a], Joel Saltz[a]

[a]Department of Biomedical Informatics, The Ohio State University, Columbus OH, 43210

1. Introduction

In many fields of medicine, engineering and science, the volume of data generated and processed is in the order of terabytes. Providing support for storing, accessing and analyzing these vast amount of datasets in a distributed environment is a challenge. This paper presents a compendium of run-time and compiler techniques and tools for supporting such applications. We will also provide a quick overview of two applications, oil reservoir management studies and digital imaging of pathology slides, that would benefit from such support.

Simulation-based oil reservoir management studies are an example of applications that generate and reference large volumes of simulation and experimental data. The objective is to develop complex numerical models of subsurface reservoirs and use these models to efficiently search for alternative oil production strategies in order to optimize profits and minimize adverse effects to the environment [35,30,41,48]. In this optimization process, there is a need to provide support for management and querying of large volumes of data generated by simulations or collected from field measurements to refine model parameters and determine the next set of simulations to be carried out. In addition, the datasets can be generated and stored at multiple locations, since the computational requirements of the simulations may require use of machines at supercomputing centers.

Digital imaging of pathology slides have gained an increasing interest over the last decade as hardware for digitizing tissue samples and microscopy slides has rapidly advanced [28]. A main challenge is storing and accessing the very large volumes of data required to represent a large collection of slides [16]. With high resolution scanners, a single focal plane of a digitized slide can be $100K \times 100K$ pixels (30 Gigabytes). Another challenging problem is the querying and processing of large volumes of data for analysis [46,47]. Analysis operations on digital slides range from simple 2D visualization and browsing of images to queries to calculate density distributions to extraction of features representing different layers of tissue or cell types to 3D reconstruction from multiple 2D scans. High performance machines are required to handle the complexity of operations and the large volume of data.

There has been considerable progress in Grid computing technologies in recent years. In addition to a wide array of middleware systems and tools, a services-based view of the Grid has emerged. In this view, data sources and applications are exposed to the environment using standard interfaces. Users interact with the resources through well-defined Grid services protocols. In this way, the complexities and heterogeneity of individual resources can be hidden from clients and greater interoperability among applications can be achieved.

*This research was supported in part by the National Science Foundation under Grants #ACI-9619020 (UC Subcontract #10152408), #EIA-0121177, #ACI-0203846, #ACI-0130437, #ANI-0330612, #ACI-9982087, #CCF-0342615, #CNS-0406386, #CNS-0426241, Lawrence Livermore National Laboratory under Grant #B517095 (UC Subcontract #10184497), NIH NIBIB BISTI #P20EB000591, Ohio Board of Regents BRTTC #BRTT02-0003.

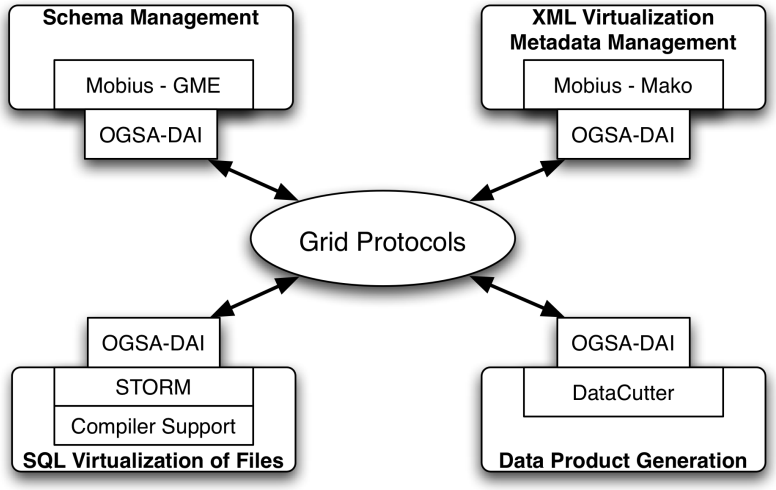


Figure 1. Middleware components and toolkits to support data-driven scientific applications in the Grid.

Several core functionalities need to be supported in an end-to-end system for enabling data-driven scientific applications in a Grid environment. These include management of data types and metadata, virtualization of data sources and data subsetting, data product generation (e.g., data aggregates from data subsets) and Grid services interfaces. In our group, we develop an integrated suite of middleware components and compiler techniques to provide these functionalities. These middleware components are shown in Figure 1. In this suite, DataCutter, which is a component-based middleware, enables combined use of task and data parallelism and is used to support data product generation (e.g., aggregates of data subsets) [13]; STORM [36,37] provides virtualization of file based datasets as object-relational tables and support for data subsetting; Mobius [27] supports management of data definitions and data types as XML schemas, XML virtualization of data, and metadata management. In an ongoing project [38], we are integrating these middleware systems with the Open Grid Services Architecture Data Access and Integration (OGSA-DAI) middleware toolkit [39] to allow access to the functionality provided by these components via OGSA-DAI Grid services protocols.

In this paper, we present a compendium of these run-time and compiler techniques and tools for supporting large scale, data-driven applications on the Grid, using two motivating applications from two different domains; oil reservoir management and digital microscopy imaging.

2. Applications

2.1. Oil Reservoir Management Studies

Effective oil reservoir management requires accurate characterization of the reservoir properties and efficient management strategies that involve optimized placement of production and injection wells. Simulation-based oil reservoir management is a viable approach to evaluate different optimization strategies and to understand changes in reservoir properties over long periods of time [30]. Various production strategies (i.e., the number and placement of injection and production wells) are simulated using a numerical model of the reservoir under study. In addition, changes in reservoir characteristics (e.g., rock properties) over time are tracked by seismic data simulations (or seismic measurements in the field). Data obtained from seismic and reservoir simulations are stored for analysis. The data analysis processes subsets of seismic simulation datasets and reservoir simulation

datasets in order to generate summary data such as production rates over a time period, bypassed oil regions in the reservoir, and rock properties in the reservoir. The results of the analysis can be used to refine the reservoir models, simulate new production strategies and collect additional seismic data.

A good understanding of fluid and rock properties in an oil reservoir is necessary for designing optimized production strategies. Since only a partial knowledge of critical parameters such as rock permeability in the reservoir is available, it is desirable to incorporate geologic uncertainty in complex reservoir models. An approach is to simulate alternative production strategies with varying number, type, timing and location of wells, applied to multiple realizations (simulation runs) of geostatistical models. This approach can lead to large volumes of output data [48].

Simulations are performed on a three-dimensional mesh over several time steps. Each realization corresponds to different geostatistical models and different number of wells and well placements. At each time step, the value of seventeen separate variables and cell locations in 3-dimensional space are output for each cell in the grid. Common analysis scenarios involve queries for economic evaluation as well as technical evaluation, such as determination of representative realizations and identification of areas of bypassed oil. Examples of client requests include *"Find all the potential bypassed oil cells between time T_1 and T_2 in realization A."* and *"Retrieve the oil saturation values at all mesh points from realizations A and B between time steps T_1 and T_2; visualize the results."*.

The physical characteristics of a reservoir change over time. These changes in reservoir material properties should be detected and incorporated into reservoir models. Seismic surveys of the reservoir can be used to track changes [30]. Seismic data is recorded as sound traces generated by multiple sound sources on the surface and sampled by receivers at the bottom and on the surface of the reservoir. The sound traces are used to infer subsurface material properties. The surveys can be either carried out in the field or simulated using the seismic models of the reservoir.

A seismic dataset is stored in files in a standard exchange format, referred to as SEGY, defined by the Society of Exploration Geophysics. A seismic data file consists of a 3600-byte header followed by a record for each sound trace. Each record contains a 240-byte header and the sound trace. The header information stores the metadata associated with the sound trace including sound source id, receiver id, receiver location, the number of samples stored for the trace. Traces collected for a single sound source are usually stored in a single file. When numerical models are used to generate seismic data, each data file can be up to 25 Gigabytes in size and there can be thousands of data sources simulated, resulting in datasets ranging from a few terabytes to hundreds of terabytes in size. We currently have about 8TB seismic simulation data on a large storage system at Ohio Supercomputer Center, 35TB on multiple TeraGrid sites, and continue to generate additional datasets.

Seismic data can be used in creating subsurface images and predicted subsurface material properties [59]. The reservoir model can be revised by imaging and inversion of output from seismic data simulations. Imaging analysis requires that subsets of seismic data be selected based on, for example, the type of sensor in a recording array and for each of a suite of sources.

2.2. Digital Microscopy Imaging

High resolution digitized microscopy enables analysis of digital mouse placenta slides to carry out quantitative examination of phenotypes and measurements of tissue structure within the placenta. One of the important components of this process is the segmentation of each image into regions of tissue layers. In our earlier work [46], we describe an approach to segment the labyrinth layer in the placenta from the rest of the image. This algorithm is based on the observation that the labyrinth layer (1) has a higher red blood cell (RBC) density compared to the spongiotropoblast and glycogen layers and (2) is elongated in shape, forming a confined strip in the image.

The algorithm consist of mainly four data processing steps: 1) *RedFinder* detects red pixels that belong to red blood cell (RBC), 2) *Counter* identifies red pixels in the regions with high RBC density, 3) *Histogram Aggregation* collects all local histograms and aggregates them into the global histogram, 4) *PCA* computes the principal direction of the high RBC density regions, and determine the bounding box for the labyrinth layer. The steps of the algorithm algorithm has been has been implemented as pipelined data processing operations using DataCutter and Mobius frameworks [46].

Mouse brain offers a convenient model for studying neuroscience. The mouse branch of BIRN [15] aims to create an integrated anatomical atlas, which provides spatial correlation for datasets ranging from CT, MR, bright-field microscopy, confocal microscopy, as well as molecular data such as genomic and protenomic data. The atlas also serves as a visual query system.

One of the challenges in creating and integrating such an atlas is in the amount of data one must work with. For example, a single image generated by a 2-photon confocal microscope can be up to 500GB. The microscope generates data as tiled images that are then stitched as a three-dimensional stack of image mosaics. While the individual image tiles are small (512 by 480 pixels), a whole image for a single z-plane is far larger. Current capability of the scanner and the amount of data it generates has already exceeded the capability of the existing software tools for stitching the images together.

The vast size necessitates distributed tools that allow parallel image stitching, volumetric visualization and analysis operations to occur in a Grid environment. In a recent work [47], we have developed mechanism for efficient preprocessing of very large digitized microscopy images on PC clusters to support efficient evaluation of a class of aggregation queries. The target class of queries involve Sum or Count operations on image pixels and pixel values. These queries are useful in computing the densities of various image attributes (e.g., red or blue pixels) in a given region of interest. The results of the queries can be used to detect and classify features in large images and segment the images. We develop a task- and data-parallel implementation of the operations using a component-based runtime system. This implementation enables pipelined processing of data through task parallelism and effective use of distributed memory and computing capacity through data parallelism. A key operation in preprocessing is the 2-dimensional(2D) prefix sum operation. We have proposed a *local 2D prefix sum* approach which eliminates the serialization and interprocessor communication overheads of a parallel full 2D prefix sum. However, this approach introduces extra operations during query evaluation. We have investigated under what conditions local 2D prefix sum is preferable to full 2D prefix sum. All the implementation builds on top of DataCutter framework.

3. Run-time System Support

Figure 1 depicts the overall system architecture and how the middleware components are interacting with each other. If we take the oil reservoir management application, in this architecture, Mobius can be used to support management of metadata associated with simulation runs, seismic field measurements, and analysis results. The structure of data types can be managed by the Mobius Global Model Exchange (GME). The support for data product generation (e.g., reconstruction of 3D volumes from seismic data, visualization of reservoir results) is provided by DataCutter. The STORM middleware can be used to support SQL-style select queries against large simulation datasets stored in distributed collection of files on a cluster of storage nodes. These components can be exposed to the Grid environment via OGSA-DAI service interfaces and can be accessed by clients using Grid Service protocols. In this section, we present brief summary of the middleware components involved in this architecture.

SELECT $< Data\ Elements >$
FROM $Dataset_1, Dataset_2, ..., Dataset_n$
WHERE $< Expression >$ AND $< Filter(< Data\ Element >) >$
GROUP-BY-PROCESSOR $ComputeAttribute(< Data\ Element >)$

Figure 2. Database queries supported by STORM.

3.1. OGSA-DAI

The Grid has emerged as an integrated infrastructure for distributed computation [21,23]. The Open Grid Services Architecture (OGSA) [22] defines mechanisms for creating, managing, and exchanging information among entities called Grid services. The objective of the OGSA-DAI [39] initiative is to build upon the OGSA infrastructure to deliver high level data management functionality for the Grid. It defines services and interfaces that can be used by clients to specify operations on data resources and data. OGSA-DAI services can be configured and customized to expose a specific database management system.

3.2. DataCutter

DataCutter [12,14,11,13] is a component-based middleware framework [40,43,29,1,5,2,17] designed to support coarse-grain data-flow execution on heterogeneous environments. In DataCutter, application processing structure is implemented as a set of components, referred to as *filters*, that exchange data through *logical streams*. A *stream* denotes a uni-directional data flow from one filter (i.e., the producer) to another (i.e., the consumer). A filter is required to read data from its input streams and write data to its output streams only. The DataCutter runtime system supports both data- and task-parallelism. Processing, network and data copying overheads are minimized by the ability to place filters on different platforms. The filtering service of DataCutter performs all steps necessary to instantiate filters on the desired hosts, to connect all logical endpoints, and to call the filter's interface functions for processing work. Data exchange between two filters on the same host is carried out by memory copy operations, while a message passing communication layer (e.g. TCP sockets or MPI) is used for communication between filters on different hosts.

3.3. STORM

STORM [36,37] is a services-oriented framework designed to support processing of large datasets in a distributed environment. It provides basic database support for 1) *selection of the data of interest* from scientific datasets stored in files and 2) *transfer of selected data from storage nodes to compute nodes for processing*. The current implementation is based on a component infrastructure, called DataCutter [13], which supports distributed execution of networks of application-specific data processing components. Using the DataCutter runtime support, STORM can perform parallel I/O on distributed data and execute data selection and data filtering operations on heterogeneous collections of storage and compute clusters.

In order to support data subsetting on file-based datasets, STORM implements three abstractions: *virtual tables*, *select queries*, and *distributed arrays*. The first two abstractions are based on object-relational database models [51]. *SELECT* operation of the form shown in Figure 2 are supported on virtual tables. Data elements selected by the *SELECT* operation are grouped based on a computed attribute. In the figure, the $< Expression >$ statement can contain value-based selections and range queries. $Filter$ allows implementation of user-defined operations that are difficult to express with simple comparison operations.

The client program that carries out data processing can be a parallel program. In that case, the distribution among client nodes of the data elements returned as the result of the query can be rep-

resented as a *distributed array*. This abstraction is incorporated into the STORM framework by the *GROUP-BY-PROCESSOR* operation in the query formulation. $ComputeAttribute$ is another user-defined function that generates the attribute value on which the selected data elements are grouped together based on the application specific partitioning of data elements.

3.4. Mobius

Mobius is a middleware framework [32,27,31] designed for efficient metadata and data management in dynamic, distributed environments. It provides a set of generic services and protocols to support distributed creation, versioning, management of database schemas, on-demand creation of databases, federation of existing databases, and querying of data in a distributed environment. Its services employ XML schemas to represent metadata definitions and XML documents to represent and exchange metadata instances. The role of *Global Model Exchange* (GME) service of Mobius is to ensure distributed model evolution and integrity while providing the ability for storage, retrieval, versioning, and discovery of models of all shape, complexity, and interconnectedness in a distributed environment. *Mobius Mako* is a service that exposes data resources as XML data services through a set of well-defined interfaces based on the Mako protocol. Our current Mako implementation provides support to expose XML databases that support the XMLDB API and contains an implementation of MakoDB. MakoDB is an XML database built on top of MySQL [34]. The Mako protocol defines methods for submitting, updating, removing, and retrieving XML documents. Upon submission, Mako assigns each entity a unique identifier. Documents, or subsets of XML documents, can be retrieved by specifying their unique identifier. XML documents can be removed by specifying their unique identifier or by specifying an element identifying XPath [9] expression. XML documents that reside in a Mako can be updated using XUpdate[2].

4. Compiler Support

Scientific datasets are typically stored as binary or character flat-files. Such *low-level* layouts enable compact storage and efficient processing. Because the use of relational or other database technologies can result in significant storage overheads and slower processing, they have typically not been very popular in most scientific communities.

The use of low-level and specialized data formats, however, makes the specification of processing much harder. Recognizing this, several ongoing projects, such as BinX and Binary Format Description (BFD) [7], are proposing machine-interpretable descriptions of binary data layouts. Data Format Definition Language (DFDL) working group under the Global Grid Forum (GGF) is trying to standardize such efforts. While such proposals can allow precise description of the datasets in a remote repository, they do not alleviate the need for detailed understanding of the formats, or the dependence of an application on a particular low-level data layout.

Using data virtualization and data services, low-level, compact, and/or specialized data formats can be hidden from the applications analyzing grid-based datasets. However, supporting data virtualization can require significant effort. For each dataset layout and abstract view that is desired, a set of data services need to be implemented. An additional challenge arises from the fact that the design and implementation of efficient data virtualization and data services oftentimes require interaction of two complementary players. The first player is the scientist who possesses a good understanding of the application, datasets, and their format, but is less knowledgeable about database and data services implementation. The second player is the database developer who is proficient in the tools and

[2]http://www.xmldb.org

techniques for efficient database and data services implementation, but has little knowledge of the specific application.

In [55], we proposed a meta-data and compiler-oriented approach to facilitate a common meeting ground for the two players and to enable automatic creation of efficient data services to support data virtualization. Specifically, we showed how a relational table like data abstraction can be supported for complex multi-dimensional scientific datasets that are resident on a cluster. By using a well-defined meta-data description language, the scientist and database developer together can describe the format of the datasets generated and used by the application. Using a compiler that can parse the meta-data description and generate code to navigate the datasets, the database developer (or the scientist) can conveniently generate data services that will serve the datasets.

In [56] we have extended our work for executing SQL-3 queries over scientific data stored as flat files. The class of queries we consider involve retrieval using Select and Where clauses, and processing with user-defined aggregate functions and group-bys.

In a more recent work [57], compiler and runtime approach has been also applied for efficient execution of multi-dimensional range queries when partial replicas of a dataset exist. A compile-time query planning strategy has been presented to select the best combination of replicas in order to minimize query execution time in a distributed environment.

5. Related Work

Grid-technologies have been employed in several large-scale, multi-institutional projects in a wide range of science and engineering domains [6,18,19,26]. GriPhyN [26] targets storage, cataloging and retrieval of large, measured datasets from large scale physical experiments. The goal is to deliver data products generated from these datasets to physicists across a wide-area network. The objective of Earth Systems Grid (ESG) [18] is to provide Grid technologies for storage, publishing, and movement of large scale data from climate modeling applications. The EUROGRID project [19] intends to develop tools for easy and seamless access to High Performance Computing (HPC) resources. The BioGrid component of the project implements the support for a uniform interface that will allow biologists and chemists to submit work to HPC facilities without having to worry about the details of running their work on different architectures. Biomedical Informatics Research Network (BIRN) (http://www.nbirn.net) [42] initiative focuses on support for collaborative access to and analysis of datasets generated by neuroimaging studies. The BIRN project uses the Storage Resource Broker (SRB) [44], which provides a distributed file system infrastructure, as a distributed data management middleware layer. MammoGrid [4] is a multi-institutional project funded by the European Union (EU). The objective of this project is to apply Grid middleware and tools to build a distributed database of mammograms and to investigate how it can be used to facilitate collaboration between researchers and clinicians across the EU. eDiamond [50] targets deployment of Grid infrastructure to manage, share, and analyze annotated mammograms captured and stored at multiple sites. One of the goals of MammoGrid and eDiamond is to develop and promote standardization in medical image databases for mammography and other cancer diseases. MEDIGRID [33,53] is another multi-institutional project investigating the application of Grid technologies for manipulating large medical image databases.

These large scale, multi-institutional projects share the same goal of deploying an infrastructure, building on Grid technologies, to facilitate sharing of data across institutions. In order to harness the collective power of distributed systems in a Grid environment, an array of tools and frameworks have been developed to support distributed storage, data replication, data processing, monitoring, security, and high-speed data transfers in Data and Computation Grids [25,24,44,58,13,3,52,54,10,27]. As

Grid computing has become more ubiquitous, an Open Grid Services Architecture (OGSA) [20,22] has been proposed. There are some recent efforts to develop Grid and Web services implementations of database technologies. Raman et. al. [45] discusses a number of *virtualization* services to make data management and access transparent to Grid applications. These services provide support for access to distributed datasets, dynamic discovery of data sources, and collaboration. Bell et. al. [8] develop uniform web services interfaces, data and security models for relational databases. Smith et. al. [49] address the distributed execution of queries in a Grid environment. They describe an object-oriented database prototype running on MPICH-G and Globus.

References

[1] The ABACUS project. *http://www.cs.cmu.edu/∼amiri/abacus.html*.

[2] Martin Aeschlimann, Peter Dinda, Julio Lopez, Bruce Lowekamp, Loukas Kallivokas, and David O'Hallaron. Preliminary report on the design of a framework for distributed visualization. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, pages 1833–1839, Las Vegas, NV, June 1999.

[3] W. Allcock, A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The Data Grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of Network and Computer Applications*, 23:187–200, 2001.

[4] R. Amendolia, F. Estrella, T. Hauer, D. Manset, D. McCabe, R. McClatchey, M. Odeh, T. Reading, D. Rogulin, D. Schottlander, and T. Solomonides. Grid databases for shared image analysis in the mammogrid project. In *The Eighth International Database Engineering & Applications Symposium (Ideas'04)*, July 2004.

[5] Khalil Amiri, David Petrou, Gregory R. Ganger, and Garth A. Gibson. Dynamic function placement for data-intensive cluster computing. In *the USENIX Annual Technical Conference*, San Diego, CA, June 2000.

[6] Asia Pacific BioGrid. http://www.apgrid.org.

[7] R. Baxter, R. Carroll, D. J. Ecklund, B. Gibbins, D̃. Virdee, and Q. Wen. BinX - A Tool for Retreiving, Searching, and Transforming Str uctured Binary Files. Available at http://www.edikit.org/binx/pub.htm, 2003.

[8] William H. Bell, Diana Bosio, Wolfgang Hoschek, Peter Kunszt, Gavin McCance, and Mika Silander. Project spitfite - towards grid web service databases. http://www.cs.man.ac.uk/grid-db/documents.html.

[9] Anders Berglund, Scott Boag (XSL WG), Don Chamberlin, Mary F. Fernndez, Michael Kay, Jonathan Robie, and Jrme Simon. *XML Path Language (XPath)*. World Wide Web Consortium (W3C), 1st edition, August 2003.

[10] Francine Berman, Andrew Chien, Keith Cooper, Jack Dongarra, Ian Foster, Dennis Gannon, Lennart Johnsson, Ken Kennedy, Carl Kesselman, John Mellor-Crummey, Dan Reed, Linda Torczon, and Rich Wolski. The GrADS Project: Software support for high-level Grid application development. *The International Journal of High Performance Computing Applications*, 15(4):327–344, November 2001.

[11] M. Beynon, T. Kurc, A. Sussman, and J. Saltz. Design of a framework for data-intensive wide-area applications. In *Proceedings of the 9th Heterogeneous Computing Workshop (HCW2000)*, pages 116–130. IEEE Computer Society Press, May 2000.

[12] Michael D. Beynon, Renato Ferreira, Tahsin Kurc, Alan Sussman, and Joel Saltz. DataCutter: Middleware for filtering very large scientific datasets on archival storage systems. In *Proceedings of the Eighth Goddard Conference on Mass Storage Systems and Technologies/17th IEEE Symposium on Mass Storage Systems*, pages 119–133. National Aeronautics and Space Administration, March 2000. NASA/CP 2000-209888.

[13] Michael D. Beynon, Tahsin Kurc, Umit Catalyurek, Chialin Chang, Alan Sussman, and Joel Saltz. Distributed processing of very large datasets with DataCutter. *Parallel Computing*, 27(11):1457–1478,

October 2001.

[14] Michael D. Beynon, Tahsin Kurc, Alan Sussman, and Joel Saltz. Optimizing execution of component-based applications using group instances. In *Proceedings of CCGrid2001: IEEE International Symposium on Cluster Computing and the Grid*, pages 56–63. IEEE Computer Society Press, May 2001.

[15] Biomedical Informatics Research Network (BIRN). http://www.nbirn.net.

[16] Umit Catalyurek, Michael D. Beynon, Chialin Chang, Tahsin Kurc, Alan Sussman, and Joel Saltz. The virtual microscope. *IEEE Transactions on Information Technology in BioMedicine*, 7(4):230–248, Dec 2003.

[17] Common Component Architecture Forum. *http://www.cca-forum.org*.

[18] Earth Systems Grid (ESG). http://www.earthsystemgrid.org.

[19] EUROGRID. http://www.eurogrid.org/.

[20] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. Grid services for distributed system integration. *IEEE Computer*, 36(6):37–46, June 2002. Open Grid Services Architecture (OGSA).

[21] Ian Foster and Carl Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, San Francisco, CA, USA, second edition, 2003.

[22] Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke. The physiology of the Grid: An open grid services architecture for distributed systems integration. http://www.globus.org/research/papers/ogsa.pdf, 2002.

[23] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the Grid: Enabling scalable virtual organization. *The International Journal of High Performance Computing Applications*, 15(3):200–222, Fall 2001.

[24] James Frey, Todd Tannenbaum, Ian Foster, Miron Livny, and Steven Tuecke. Condor-G: A computation management agent for multi-institutional grids. In *Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing (HPDC10)*. IEEE Press, Aug 2001.

[25] The Globus Project. http://www.globus.org.

[26] Grid Physics Network (GriPhyN). http://www.griphyn.org.

[27] Shannon Hastings, Stephen Langella, Scott Oster, and Joel Saltz. Distributed data management and integration: The mobius project. In *GGF Semantic Grid Workshop 2004*, pages 20–38. GGF, June 2004.

[28] Interscope technologies. *http://www.interscopetech.com*, 2001.

[29] Carsten Isert and Karsten Schwan. ACDS: Adapting computational data streams for high performance. In *14th International Parallel & Distributed Processing Symposium (IPDPS 2000)*, pages 641–646, Cancun, Mexico, May 2000.

[30] T. Kurc, U. Catalyurek, X. Zhang, J. Saltz, R. Martino, M. Wheeler, M. Peszyńska, A. Sussman, C. Hansen, M. Sen, R. Seifoullaev, P. Stoffa, C. Torres-Verdin, and M. Parashar. A simulation and data analysis system for large scale,data-driven oil reservoir simulation studies. *Concurrency and Computation: Practice and Experience.*, 17(11):1441–1467, September 2005.

[31] Stephen Langella, Shannon Hastings, Scott Oster, Tahsin Kurc, Umit Catalyurek, and Joel Saltz. A distributed data management middleware for data-driven application systems. In *Proceedings of 2004 IEEE International Conference on Cluster Computing*, September 2004.

[32] The Mobius Project. http://www.projectmobius.org.

[33] J. Montagnat, V. Breton, and I. E. Magnin. Using grid technologies to face medical image analysis challenges. In *BioGrid'03, The 3rd International Symposium on Cluster Computing and the Grid (CCGrid 2003)*, pages 588–593, May 2003.

[34] MySql Database. *http://www.mysql.com/*.

[35] Sivaramakrishnan Narayanan, Umit Catalyurek, Tahsin Kurc, Vijay S Kumar, and Joel Saltz. A runtime framework for partial replication and its application for on-demand data exploration. In *High Performance Computing Symposium (HPC 2005), SCS Spring Simulation Multiconference*, Mar 2005.

[36] Sivaramakrishnan Narayanan, Umit Catalyurek, Tahsin Kurc, Xi Zhang, and Joel Saltz. Applying database support for large scale data driven science in distributed environments. In *Proceedings of*

the Fourth International Workshop on Grid Computing (Grid 2003), pages 141–148, Phoenix, Arizona, Nov 2003.

[37] Sivaramakrishnan Narayanan, Tahsin Kurc, Umit Catalyurek, and Joel Saltz. Database support for data-driven scientific applications in the grid. *Parallel Processing Letters*, 13(2):245–271, 2003.

[38] Sivaramakrishnan Narayanan, Tahsin M. Kurc, Umit V. Catalyurek, and Joel H. Saltz. Servicing seismic and oil reservoir simulation data through grid data services. In *Proceedings of VLDB Workshop Data Management in Grid 2005 (VLDB DMG'05*, pages 98–109, Sep 2005.

[39] Open Grid Services Architecture Data Access and Integration (OGSA-DAI). http://www.ogsadai.org.uk.

[40] Ron Oldfield and David Kotz. Armada: A parallel file system for computational grids. In *Proceedings of CCGrid2001: IEEE International Symposium on Cluster Computing and the Grid*, Brisbane, Australia, May 2001. IEEE Computer Society Press.

[41] Manish Parashar, Vincent Matossian, Wolfgang Bangerth, Hector Klie, Benjamin Rutt, Tahsin M. Kurç, Ümit V. Catalyurek, Joel H. Saltz, and Mary F. Wheeler. Towards dynamic data-driven optimization of oil well placement. In Vaidy S. Sunderam, G. Dick van Albada, Peter M. A. Sloot, and Jack Dongarra, editors, *International Conference on Computational Science (2)*, volume 3515 of *Lecture Notes in Computer Science*, pages 656–663. Springer, 2005.

[42] S.T. Peltier and M.H. Ellisman. *The Biomedical Informatics Research Network, in* The Grid, Blueprint for a New Computing Infrastructure. 2nd Edition: Elsevier, 2003.

[43] Beth Plale and Karsten Schwan. dQUOB: Managing large data flows using dynamic embedded queries. In *IEEE International High Performance Distributed Computing (HPDC)*, August 2000.

[44] Arcot Rajasekar, Michael Wan, and Reagan Moore. MySRB & SRB - components of a data grid. In *The 11th International Symposium on High Performance Distributed Computing (HPDC-11)*, July 2002.

[45] Vijayshanker Raman, Inderpal Narang, Chris Crone, Laura Haas, Susan Malaika, Tina Mukai, Dan Wolfson, and Chaitan Baru. Data access and management services on grid. http://www.cs.man.ac.uk/grid-db/documents.html.

[46] M. Ribeiro, T. Kurc, T. Pan, K. Huang, U. Catalyurek, X. Zhang, S. Langella, S. Hastings, S. Oster, R. Ferreira, and J. Saltz. *Grid Computing: The New Frontier Of High Performance Computing, 14*, chapter Tools for Efficient Subsetting and Pipelined Processing of Large Scale, Distributed Biomedical Image Data. Elsevier, 2005.

[47] Benjamin Rutt, Vijay S. Kumar, Tony Pan, Tahsin Kurc, Umit Catalyurek, Yujun Wang, and Joel Saltz. Distributed out-of-core preprocessing of very large microscopy images for efficient querying. In *The 2005 IEEE International Conference on Cluster Computing*, Boston, MA, Sep 2005.

[48] J. Saltz, U. Catalyurek, T. Kurc, M. Gray, S. Hastings, S. Langella, S. Narayanan, R. Martino, S. Bryant, M. Peszynska, M. Wheeler, A. Sussman, M. Beynon, C. Hansen, D. Stredney, , and D. Sessanna. Driving scientific applications by data in distributed environments. In *Dynamic Data Driven Application Systems Workshop, held jointly with ICCS 2003*, Melbourne, Australia, June 2003.

[49] Jim Smith, Anastasios Gounaris, Paul Watson, Norman W. Paton, Alvaro A.A. Fernandes, and Rizos Sakellariou. Distributed query processing on the grid. http://www.cs.man.ac.uk/grid-db/documents.html.

[50] A. Solomonides, R. McClatchey, M. Odeh, M. Brady, M. Mulet-Parada, D. Schottlander, and S.R Amendolia. Mammogrid and ediamond: Grids applications in mammogram analysis. In *Proceedings of the IADIS International Conference: e-Society 2003*, pages 1032–1033, 2003.

[51] Michael Stonebraker and Paul Brown. *Object-Relational DBMSs, Tracking the Next Great Wave*. Morgan Kaufman Publishers, Inc., 1998.

[52] D. Thain, J. Basney, S. Son, and Miron Livny. Kangaroo approach to data movement on the grid. In *Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing (HPDC10)*, 2001.

[53] T. Tweed and S. Miguet. Medical image database on the grid: Strategies for data distribution. In *HealthGrid'03*, pages 152–162, January 2003.

[54] S. Vazhkudai, S. Tuecke, and I. Foster. Replica selection in the globus data grid. In *International Workshop on Data Models and Databases on Clusters and the Grid (DataGrid 2001)*. IEEE Computer Society Press, 2001.

[55] Li Weng, Gagan Agrawal, Umit Catalyurek, Tahsin Kurc, Sivaramakrishnan Narayanan, and Joel Saltz. An approach for automatic data virtualization. In *Proceedings of the Thirteen International Symposium on High Performance Distributed Computing (HPDC-13)*, Honolulu, HI, Jun 2004. IEEE Press.

[56] Li Weng, Gagan Agrawal, Umit Catalyurek, and Joel Saltz. Supporting sql-3 aggregations on grid-based data repositories. In *Proceedings of the 17th International Workshop on Languages and Compilers for Parallel Computing*, 2004.

[57] Li Weng, Umit Catalyurek, Tahsin Kurc, Gagan Agrawal, and Joel Saltz. Servicing range queries on multidimensional datasets with partial replicas. In *Proceedings of the 5th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2005)*, May 2005.

[58] R. Wolski, N. Spring, and J. Hayes. The network weather service: A distributed resource performance forecasting service for metacomputing. *Journal of Future Generation Computing Systems*, 15(5-6):757–768, 1999.

[59] Xi Zhang, Benjamin Rutt, Umit Catalyurek, Tahsin Kurc, and Joel Saltz. Supporting scalable and distributed data subsetting and aggregation in large-scale seismic data analysis. *The Journal of High Performance Computing Applications*, 2006. to appear.

Periscope: Advanced Techniques for Performance Analysis*

Michael Gerndt [a], Karl Fürlinger[a], Edmond Kereku[a]

[a]Technische Universität München, Fakultät für Informatik 10, Email: gerndt@in.tum.de

Performance analysis of applications on supercomputers require scalable tools. The Periscope environment applies a distributed automatic online analysis and thus scales to thousands of processors. This article gives an overview of the Periscope system, from the performance property specification, via the search process, to the integration with two monitoring systems. We also present first experimental results.

1. Introduction

Supercomputers with up to hundreds of teraflops are currently build and deployed to solve grand challenge applications. These systems consist of thousands of processors arranged in a clustered SMP architecture, i.e., the basic building block is an SMP system with a small number of processors. Applications executed on such systems should be as efficient as possible due to the enormous costs of the machines and due to the goal of enabling new research results through high performance computing.

A major step in developing applications for such machines is therefore performance analysis and tuning. Optimizing an application for best performance already in the design phase is almost impossible due to the complex architecture of such machines. Actual program runs with real data and the desired number of processors have to be analyzed to detect performance problems and their causes. Scaling down the application while keeping the performance characteristics unchanged is almost impossible.

Our Periscope performance analysis environment supports the programmer in analyzing the performance of full size application runs. The major challenge solved by the tool is scalability to a large number of processors. This is possible by performing an automatic distributed online analysis. Analysis agents are distributed across all SMP nodes assigned to the application. Each agent automatically searches for performance problems of the processes and threads running on the SMP node. Low-level performance data are analyzed locally and detected performance problems are communicated through the network to the master agent which interacts with the programmer.

Periscope is based on a set of techniques developed in the APART working group on automatic performance analysis tools. The potential performance problems are formalized in ASL (APART Specification Language) [2] [3] and the interface between the monitoring system and the node agents is an extension of MRI (Monitoring Request Interface) [7]. The agents take into account also the program's structure, e.g., the nesting of program regions. This information is provided in SIR (Standard Intermediate Program Representation) [14].

This article gives an overview of Periscope and presents first results from real experiments. Section 2 outlines related work and Section 3 the APART Specification Language. The overall design of the Periscope system is presented in Section 4. It covers also the integration of different monitoring systems into Periscope. In the center of Periscope are the search strategies which are described in Section 5. Section 6 gives a usage scenario and Section 7 presents actual results from applying Periscope to the NAS Parallel Benchmarks and an application code. The article closes with a short

*This work is funded by the German Science Foundation (DFG).

```
PROPERTY LC1ReadMissesOverMemRef(SeqPerf s){
CONDITION:
  s.lc1_data_read_miss/s.read_access > 0.01;
CONFIDENCE:
  1;
SEVERITY:
  s.lc1_data_read_miss/s.read_access; }
```

Figure 1. This performance property identifies significant L1-cache miss rate.

summary and outlook in Section 8.

2. Related Work

Several projects in the performance tools community are concerned with the automation of the performance analysis process. Paradyn's [11] Performance Consultant automatically searches for performance bottlenecks in a running application by using a dynamic instrumentation approach. Based on hypotheses about potential performance problems, measurement probes are inserted into the running program. Recently MRNet [13] has been developed for the efficient collection of distributed performance data. However, the search process for performance data is still centralized.

The Expert [15] tool developed at Forschungszentrum Jülich performs an automated post-mortem search for patterns of inefficient program execution in event traces. Potential problems with this approach are large data sets and long analysis times for long-running applications that hinder the application of this approach on larger parallel machines.

Aksum [1], developed at the University of Vienna, is based on a source code instrumentation to capture profile-based performance data which is stored in a relational database. The data is then analyzed by a tool implemented in Java that performs an automatic search for performance problems based on JavaPSL, a Java version of ASL.

3. APART Specification Language

Periscope starts its analysis from the formal specification of performance properties in the APART Specification Language (ASL) [4]. The specification determines the condition, the confidence value and the severity of performance properties. Beyond the individual specification of each property, the ASL provides property templates for specifying similar performance properties in a compact way and *metaproperties* for combining already defined properties. These techniques lead to extremely compact specifications.

The example in Figure 1 demonstrates the specification of performance properties with ASL. The performance property shown here identifies a region with high L1-cache miss rate. The data model, specified in ASL too, contains a class `SeqPerf` which contains a reference to a program region, a reference to a process, and a number of cache-related and other metrics. Figure 2 shows the C++-classes generated from the ASL specification. The instance of `SeqPerf` available in the property is called the property's context. It defines the program region, e.g., a function, and the process for which the property is tested.

The condition determines whether the property holds. It accesses information in the data model.

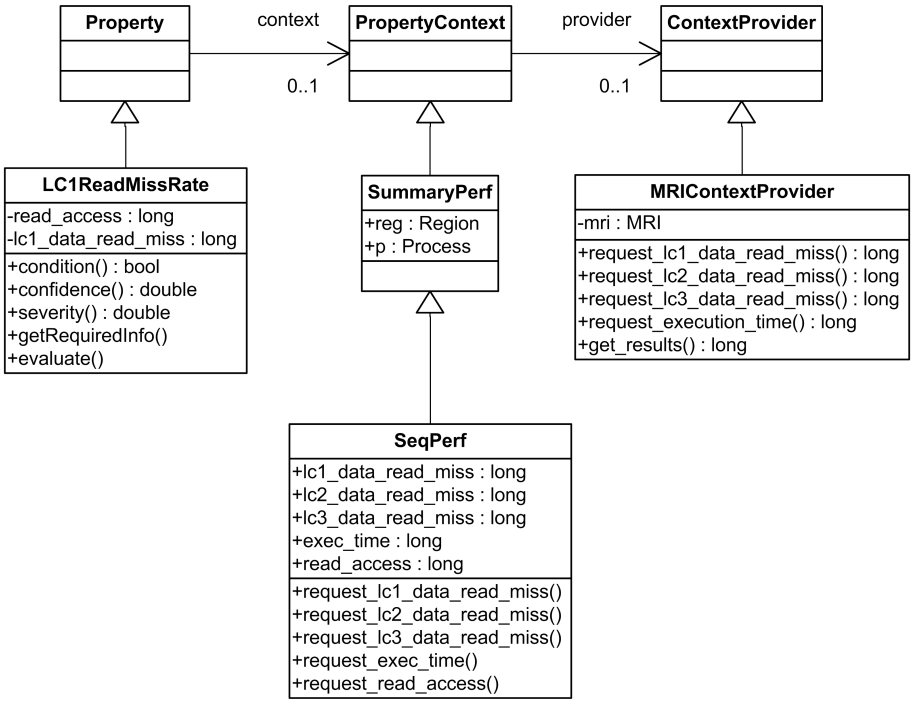


Figure 2. This figure presents the C++-classes generated from the ASL specification in Figure 1.

The confidence is 1 since it is not only a hint for the property but a proven property based on measured performance data. The severity returns the miss rate and allows to rank similar properties for all program regions and processes. Condition, confidence, and severity are implemented by methods of the C++-class generated from the ASL specification.

4. Periscope Design

Periscope consists of a graphical user interface based on Eclipse, a hierarchy of analysis agents and two separate monitoring systems (Figure 3).

The graphical user interface allows the user to start up the analysis process and to inspect the results. The agent hierarchy performs the actual analysis. The node agents autonomously search for performance problems which have been specified with ASL. Typically, a node agent is started on each SMP node of the target machine. This node agent is responsible for the processes and threads on that node. Detected performance problems are reported to the master agent that communicates with the performance cockpit.

The node agents access a performance monitoring system for obtaining the performance data required for the analysis. Periscope currently supports two different monitors, the *Peridot monitor* [5] developed in the Peridot project focusing on OpenMP and MPI performance data, and the *EP-Cache monitor* [10] developed in the EP-Cache project focusing on memory hierarchy information. Both monitors are described in more detailed in the following subsections.

The node agents perform a sequence of experiments. Each experiment lasts for a program phase, which is defined by the programmer, or for a predefined amount of execution time. Before a new experiment starts, an agent determines a new set of hypothetical performance problems based on the

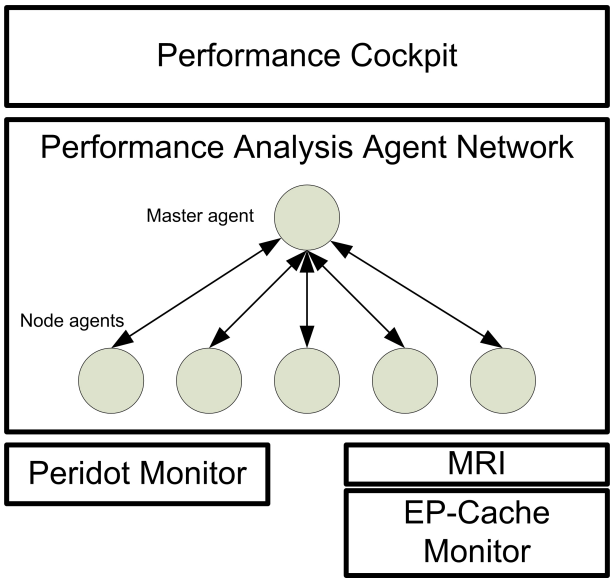


Figure 3. Periscope currently consists of a GUI based on Eclipse, a hierarchy of analysis agents, and two separate monitoring systems.

predefined ASL properties and the already found problems. It then requests the necessary performance data for proving the hypotheses and starts the experiment. After the experiment, the hypotheses are evaluated based on the performance data obtained from the monitor. Subsection 5 gives more details on the currently implemented search strategies.

4.1. Periscope Monitors

Although Periscope can be adapted for other monitoring systems, we currently use two monitors that where developed in two previous projects. The Peridot monitor includes new techniques to reduce the measurement overhead by offloading some of the tasks in monitoring to an additional monitoring process. The EP-Cache monitor reduces overhead by supporting detailed monitor configuration with the goal to measure only what is really necessary.

4.1.1. Distributed Monitoring

The first monitoring system was developed in the Peridot project funded by KONWIHR. The target system was the Hitachi SR8000 installed at LRZ München. Its building blocks are 9-way SMP systems where 8 processors are available to user programs and the ninth is used by the OS.

The Peridot Monitor is shown in Figure 4(a). The node agent accesses the *Runtime Information Producer (RIP)* which is executed on a separate processor, i.e., on the Hitachi it is the OS processor of a node. In this figure processor 1 is one of the application processors. The application is linked with a monitoring library which has very little overhead since data are efficiently placed in an circular event buffer. No control decisions or aggregation operations are executed by the monitoring library. It is the task of the RIP to perform any time consuming operations.

The event buffer is placed in shared memory of an SMP node so that all application processes can deliver their performance data and the RIP can also access the buffer. In addition to utilizing physical shared memory, the implementation also provides support for the Remote Memory Access

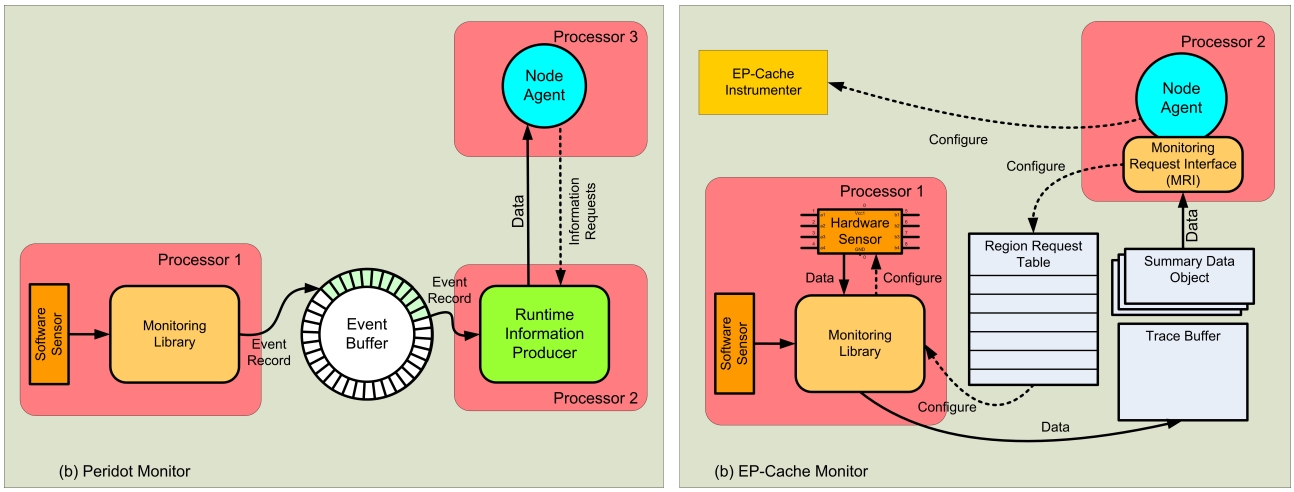


Figure 4. (a) Distributed Monitoring with the Peridot Monitor (b) Configurable Monitoring with the EP-Cache Monitor.

hardware of the Hitachi. Thus, the RIP can be executed on another SMP node and still efficiently access the event buffer.

The node agent accesses information in a pull model, thus it sends a request and receives the available performance data. Calls to the Peridot Monitoring Library are inserted by source code instrumentation with OPARI [12], via the MPI profiling interface, as well as by the function call instrumentation capability of GNU compilers (the `-finstrument-functions` compiler option).

4.2. Configurable Monitoring

The second monitoring system is the EP-Cache monitor in Figure 4(b). The goal of the EP-Cache project was to investigate new techniques for hardware monitoring of accesses to the memory hierarchy. The project developed a design of a hardware monitor that can monitor address ranges. This, for examples, allows to count only cache misses that occur for a specific array. In addition, the monitor can be configured to provide histogram information for address ranges with a predefined granularity, e.g., multiples of cache lines.

To be able to utilize this monitor's features effectively, the monitoring library had to support online configuration of the hardware monitor. For example, different data structures should be monitored in different loops of the program. This requires to reconfigure the monitor for the loop to be executed. We extented the configuration support for the hardware monitor also for software sensors that are inserted into the soure program by a source-to-source instrumentation tool called *EP-Cache Instrumenter* [8]. Since the monitoring overhead also depends on the amount of instrumentation, in principle, Periscope could also guide the instrumentation process, as shown in Figure 4(b).

To enable configuration requests for program regions, the instrumenter generates a file with information about the program structure. It utilizes the *Standard Intermediate Program Representation* (SIR) [14] developed in the APART working group. It is an XML notation for specifying - besides other information - program regions, their nesting structure, as well as the data structures accessed in the regions. This SIR file is read by Periscope so that it knows about the program regions and data structures and can use this information in the performance property search.

The node agent communicates with the EP-Cache Monitor via the *Monitoring Request Interface* (MRI) [7]. The MRI is based on a proposal in the APART group but was extended for the special needs in EP-Cache. MRI defines a set of operations to send measurement requests to the monitor. For example, the node agent can request measurement of L1 cache misses for data structure `A` in a specific loop. MRI also provides operations to retrieve the measured information for a specific request. The last functionality in MRI is to control program execution. The program can, for example, be started for the next program phase (see Section 5) by specifying the end of a program region which will automatically stop the program. Thus, the node agent can specify requests, start the execution of the next phase, retrieve and evaluate the performance data, and take decisions for the execution of the next phase (an experiment).

The requests as well as the resulting performance data, either aggregated information or trace information, are exchanged between the node agent and the application processor via a shared memory area.

4.3. Adapting Periscope to other Monitors

Currently, Periscope can make use of the Peridot and the EP-Cache monitor. While the first one allows measurements for OpenMP and MPI, the second one provides support for detailed measurements of the memory access behavior of OpenMP programs.

The two monitoring systems demonstrate that Periscope can be adapted to different monitoring systems. The following tasks are necessary to base Periscope on yet another monitor:

1. Define the ASL data model

   The ASL data model used in the specification of the performance properties (see Section 3) has to include only performance data that can be measured with the underlying monitor. Thus, the data model and probably the properties have to be adapted to the new monitor.

2. Develop a context provider

   Figure 2 shows that a property context is connected to a context provider. The context provider implements the connection to the monitor. In the figure, the `MRIContextProvider` class implements the connection to an MRI-based monitor, e.g., the EP-Cache monitor. It provides methods to request measurements as well as to access the measurement results. The same operations would have to be implemented for a class designed for another monitor, but only for those data that are included in the data model.

5. Search Strategies

The analysis agent is based on a small set of databases.

1. **Application Structure**: It stores the structure of the application which is available to Periscope through the SIR file generated by the F90 instrumenter.

2. **ASL Property**: It contains the C++-classes generated from the ASL property specification. Properties can be added to the database via shared libraries without recompilation of Periscope.

3. **Performance Data**: It stores the performance data retrieved from the monitor. It is based on the classes generated from the ASL performance model.

4. **Performance Problems**: This database stores the detected performance problems.

5. **Search Strategies**: Periscope currently provides several search strategies. The strategies determine which performance property hypotheses are evaluated in the next experiment.

As mentioned above, the whole search for performance problems happens online in a distributed fashion. The overall goal is to enable performance analysis for large scale applications.

The analysis executes a series of *experiments*. These experiments can be individual runs of the application as well as executions of subsequent phases during a single program run. Each experiment consists of several steps:

1. Hypothesis selection

   Based on the set of already detected performance problems, the structure of the application, the characteristics of the phase (cyclic of execute once), and on the limits of the monitoring system the next set of hypothesis to be tested is selected.

2. Monitor configuration

   The monitor is configured to deliver the performance data required to evaluate the performance hypotheses.

3. Experiment execution

   The application is released until it stops and the control goes back to Periscope.

4. Retrieving results

   The measured performance data need to be fetched from the monitor. The data are inserted into the performance data base, e.g., in the `SeqPerf` object. This step is implemented by the `getResults` method of the context provider.

5. Evaluation of hypothesis

   The analysis agent calls the `condition` method of the properties to check their existence. This method accesses the data in the performance model that were inserted in the previous step.

Periscope currently supports a small set of strategies for determining the performance problems. These strategies are a non-phase-based strategy and two phase-based strategies.

The *non-phase-based strategy* periodically evaluates all possible hypotheses. This strategy makes only sense in combination with the Peridot monitor, since explicit configuration is not necessary. Thus the monitor configuration step is omitted. After a predefined amount of time, the node agent retrieves the performance data and evaluates the hypotheses. In the next experiment, the set of hypotheses is adapted and evaluated after the next time interval.

The two phase-based strategies depend on the phase concept. The application is split into phases, called *application phases*. More precisely, a phase is a specific program region. The overall execution of the program thus includes a sequence of instances of those phases. Different applications are constructed with different phases. One possible example is depicted in Figure 5. Once the user identified those phases, appropriate measurements can be performed during the execution of those phases. In many of the HPC applications there is a time loop which consumes most of the execution time. This loop's body is an excellent example for a repetitive application phase.

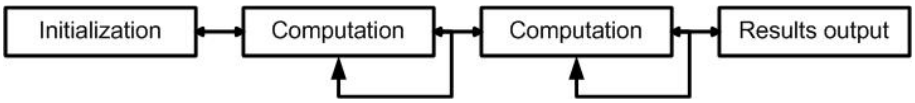


Figure 5. An example application including an initialization phase, several computational phases(which can be repetitive), and a concluding phase for possible deallocation and results gathering.

Periscope can basically use each instrumented code region as an application phase. While the phases are mainly intended to support online analysis, they can also be used to support iterative analysis via repetitive execution of the program. The phase in this case is the program's main routine.

The user may want to mark other code regions than standard code regions as application phases. Standard regions are, for example, functions, sequential loops, and parallel regions in OpenMP. We implemented a simple mechanism in our instrumenter, which allows specifying those phases everywhere in the application with very little effort. It only requires the user to insert "USER REGION" directives in the code before the instrumentation.

The first phase-based strategy gives higher priority to the identification of the location of performance problems. It first searches for basic performance properties, such as a significant L1- and L2-cache miss rate. It starts with the main program and the call sites in the main program. If a call site shows a high cache miss rate, it checks hypotheses for the outermost regions in that function. In the following experiments it checks for nested regions as well as for specific data structures that are responsible for the miss rate.

The incremental search for the location of a performance problem is required, e.g., due to the limited number of counters in the hardware monitor or due to the instrumentation overhead for fine grained program regions.

After the detection of the location, the search strategy refines in the performance properties. For example, the high miss rate can result from read or write misses, or from conflict and - in the case of multiprocessors - from coherence misses.

The second phase-based strategy gives priority to the refinement in the property hierarchy. Thus, before it refines in the regions and data structures, it will search for all possible performance problems for the regions currently under investigation.

6. Usage Scenario

To illustrate a usage of our analysis framework consider the following usage scenario, where a user wants to conduct an performance analysis of his MPI-based application based on the non-phase-based search strategy using the Peridot monitor. Note that this just represents one possible application scenario of our tool, a usage for OpenMP-based applications or using a batch system instead of interactive startup is possible too, for example.

1. Preparation of the application for performance analysis. In order to enable performance analysis of the application it has to be instrumented to enable the recording of performance data by our monitoring systems. Currently we support OpenMP monitoring based on the Opari [12] source-to-source instrumenter, MPI monitoring based on the MPI performance monitoring interface, and function-call monitoring based on the function call instrumentation capability

of GNU compilers (the `-finstrument-functions` compiler option). The instrumented application is then compiled and linked against our monitoring library.

2. Interactive startup of the application. The user starts his application using the `mpirun` command on a number of nodes of the system. The application begins to execute and on the first call into our monitoring library (typically, the `MPI_Init` call), the library performs the necessary initialization procedures like allocating buffers for the generated performance data. Furthermore the monitoring library registers itself with a registry service that is used by the components in our system to discover each other. At this point, the monitoring library halts the execution of the target application (unless a special environment variable is set) to allow the agent system to be started to monitor the execution of the application from the start.

3. The user interactively starts the performance analysis system by executing `./periscope <name of the application>`. The startup component then contacts the registry service to determine on which nodes of the system the application is executing and starts a node agent on each of the identified nodes.

4. After all node agents have been started a master agent is started on the interactive node and a `start` command is issued by the master agent to all node agents. This causes the halted application to be resumed.

5. Performance data is now generated by the target application and processed by the node agents according to the search strategy. Specifically, property contexts are instantiated as demanded for the evaluation of the properties to be checked. The `MPIOverhead` property, for example, is evaluated on `SeqPerf` contexts. Therefore, the node agents access the RIP (via the `RIPContextProvider`). The node agent can then apply the `MPIOverhead` property which in turn internally accesses the MPI data member of the `SeqPerf` class.

6. The discovered performance properties are communicated to the master agent that reports them to the user.

7. Results

As a first example, we present here the performance analysis of a crystal growth simulation code. It simulates the melting and crystallization process in the production of silicon wavers. The application consists of a time loop in which the temperature distribution and the flow of the melt is simulated. This is a sequential version of the code where Periscope searches for memory hierarchy problems.

In the first experiment, i.e., the first execution of the time loop, the call sites in the time loop are measured, i.e., calls to procedures CURR, VELO, TEMP, and BOUND. The results of the experiment are the following performance problems:

```
LC1MissesOverMemRef
  Severity: 0.156674
  Region: BOUND( CALL, main.f, 69 )
LC1MissesOverMemRef
  Severity: 0.0225174
  Region: TEMP( CALL, main.f, 68 )
```

The most severe performance problem is found in procedure BOUND which performs the bound-

aries update. In the next experiment the regions in BOUND are analyzed. This results in the following performance properties:

```
LC1MissesOverMemRef
  Severity: 0.68164
  Region: ( LOOP, bound.f, 139 )
LC1MissesOverMemRef
  Severity: 0.216865
  Region: ( LOOP, bound.f, 673 )
```

The next two experiments investigate the precise type of misses and the data structures responsible. The final list of performance problems found in that program run includes the following problems:

```
LC1WriteMissesOverMemRef (LOOP, bound.f, 139 )
  Severity: 0.671016
  Data Structure: UN
LC1MissesOverMemRef (LOOP, bound.f, 28 )
  Severity: 0.103128
  Data Structure: UN
LC1ReadMissesOverMemRef (LOOP, bound.f, 673 )
  Severity: 0.108437
LC1WriteMissesOverMemRef (LOOP, bound.f, 673 )
  Severity: 0.108429
```

The first property shows that the misses are almost all write misses due to array UN. The next property points to a problem with cache misses for data structure UN in loop 28 (This is the line number in file bound.f). But no more specific properties could be determined for that region. In contrast to the result for this loop, Periscope was able to prove that read and write misses have the same severity in loop 673 but no single data structure is responsible for the misses.

We also applied Peridot to the NAS Parallel Benchmarks, some application programs from the EP-Cache project, and the APART Test Suite [6]. Figure 6 shows the number of performance properties found for an OpenMP version of the NAS parallel benchmarks.

8. Summary and Future Work

This paper gives on overview about the Periscope system for automatic performance analysis of applications on teraflop computers. Scalability to execution runs on thousands of processors is achieved by executing an online distributed analysis. Analysis agents are autonomously searching for performance problems of a small subset of the threads or processes and reporting the problems back to a master agent.

In our experiments a single node agent is responsible for an SMP node of the target machine. Other configurations can certainly be implemented if the requirement of the communication interface with the monitoring system, i.e., communication via shared memory, is available.

This article presents also some first results applying Periscope to an application code and to the NAS Parallel Benchmarks. Periscope can search for a predefined set of performance properties. In these programs, either a database with properties for memory hierarchy problems is used or a database with performance problems for OpenMP programs.

Property	BT	CG	EP	FT	IS	LU	MG	SP
ImbalanceAtBarrier					1	3		
ImbalanceInParallel-Sections								
ImbalanceInParallelLoop	12	13	1	8	2	9	12	16
ImbalanceInParallel-Region	6	9	1		2	8	2	5
UnparallelizedInSingle-Region						3		
UnparallelizedInMaster-Region	4					13	2	5
ImbalanceDueToNotEnough-Sections								
ImbalanceDueToUneven-SectionDistribution								
CriticalSectionContention								1
LockContention								

Figure 6. This table shows the number of performance problems found in the OpenMP version of the NAS parallel benchmarks.

We do not yet provide a nice GUI for Periscope, but we are implementing a GUI based on Eclipse. It will be used for phase specification, configuration of the search process, as well as for presenting and explaining the performance problems found.

Finally the environment can be used for performance analysis of Grid applications [9]. It is important in this context too, to do local analysis and to propagate only small amounts of high level information among the analysis agents.

References

[1] T. Fahringer, C. Seragiotto: *Aksum: A Performance Analysis Tool for Parallel and Distributed Applications*, Performance Analysis and Grid Computing, Eds. V. Getov, M. Gerndt, A. Hoisi, A. Malony, B. Miller, Kluwer Academic Publisher, ISBN 1-4020-7693-2, pp. 189-210, 2003

[2] T. Fahringer, M. Gerndt, G. Riley, J.L. Träff: *Specification of Performance Problems in MPI-Programs with ASL*, International Conference on Parallel Processing (ICPP'00), pp. 51 - 58, 2000

[3] T. Fahringer, M. Gerndt, G. Riley, J.L. Träff: *Formalizing OpenMP Performance Properties with the APART Specification Language (ASL)*, International Workshop on OpenMP: Experiences and Implementation, Lecture Notes in Computer Science, Springer Verlag, Tokyo, Japan, pp. 428-439, 2000

[4] T. Fahringer, M. Gerndt, G. Riley, J. Träff: *Knowledge Specification for Automatic Performance Analysis*, APART Technical Report, www.fz-juelich.de/apart, 2001

[5] K. Fürlinger, M. Gerndt: *Peridot: Towards Automated Runtime Detection fo Performance Bottlenecks*, High Performance Computing in Science and Engineering, Garching 2004, pp. 193-202, Springer, 2005

[6] M. Gerndt, B. Mohr, J. Larsson-Traeff: *Evaluating OpenMP Performance Analysis Tools with the APART Test Suite*, Fifth European Workshop on OpenMP (EWOMP '03), RWTH Aachen, pp. 147-156, 2003

[7] M. Gerndt, E. Kereku: *Monitoring Request Interface Version 1.0*, TUM Technical Report, 2003

[8] M. Gerndt, E. Kereku: *Selective Instrumentation and Monitoring*, International Workshop on Compilers for Parallel Computers (CPC 04), 2004

[9] M. Gerndt: *Automatic Performance Analysis Tools for the Grid*, Concurrency and Computaton: Practice

and Experience, Vol. 17, pp. 99-115, 2005

[10] E. Kereku, M. Gerndt: *The EP-Cache Automatic Monitoring System*, International Conference on Parallel and Distributed Systems (PDCS 2005), 2005

[11] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, T. Newhall: *The Paradyn Parallel Performance Measurement Tool*, IEEE Computer, Vol. 28, No. 11, pp. 37-46, 1995

[12] A. Malony, B. Mohr, S. Shende, F. Wolf: *Towards a Performance Tool Interface for OpenMP: An Approach Based on Directive Rewriting*, EWOMP 01, Third European Workshop on OpenMP, 2001

[13] Ph. C. Roth, D. C. Arnold, B. P. Miller: *MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools*, SC2003, Phoenix, November, 2003

[14] C. Seragiotto, H. Truong, T. Fahringer, B. Mohr, M. Gerndt, T. Li: *Standardized Intermediate Representation for Fortran, Java, C and C++ Programs*, APART Working Group Technical Report, Institute for Software Science, University of Vienna, Octorber, 2004

[15] F. Wolf, B. Mohr: *Automatic Performance Analysis of Hybrid MPI/OpenMP Applications*, 11th Euromicro Conference on Parallel, Distributed and Network-Based Processing, pp. 13 - 22, 2003

The Mitosis Speculative Multithreaded Architecture

Carlos Madriles[a], Carlos García Quiñones[a], Jesús Sánchez[a], Pedro Marcuello[a], Antonio González[a]

[a]Intel Barcelona Research Center, Intel Labs - UPC, Barcelona

Abstract

Speculative multithreading (SpMT) increases the performance by means of exploiting speculative thread-level parallelism. In this paper we describe the Mitosis framework, which is a combined hardware-software approach to finding and exploiting speculative thread-level parallelism, even in the presence of frequent dependences between threads. The approach is based on predicting/computing thread input values via software, through a piece of code that is added at the beginning of each thread, the pre-computation slice (p-slice). A p-slice is expected to compute the correct thread input values most of the time, but not necessarily always. Because of that, aggressive optimization techniques can be applied to the slice to make it very short. In this paper, we also describe the microarchitecture that supports this execution model. The main novelty of the microarchitecture is the organization of the register file and the cache memory in order to support multiple versions of each variable and allow for roll-back in case of misspeculation. We also describe a novel compiler approach to identify points where speculative threads are most effectively spawned, and generate the corresponding p-slices for each. We show that the Mitosis microarchitecture-compiler achieves very important speedups for applications that the compiler cannot parallelize by conventional non-speculative approaches, such as the Olden benchmarks.

1. Introduction

Most high-performance processor vendors have now introduced designs that feature processors that can execute multiple threads simultaneously on the same core, either through multithreading [5][15], multiprocessing [20][22] or a combination of the two. The way thread-level parallelism (TLP) is currently being exploited in these processors is through non-speculative threading – all created threads are committed. Basically, there are two main sources of non-speculative TLP: 1) execute different applications in parallel, and 2) execute parallel threads from a single application generated through compiler/programmer support. In the former case, running different independent applications in the same processor provides an increase in throughput (number of jobs finished per time unit) and a reduction in the average response time of jobs over a single-threaded processor. In the latter case, programs are partitioned into smaller threads that are executed in parallel. This partitioning process may significantly reduce the execution time of the parallelized application in comparison with single-threaded execution.

Executing different applications in parallel provides no gain for a single application. On the other hand, partitioning applications into parallel threads may be a straightforward task in regular applications, but becomes much harder for irregular programs, where compilers usually fail to discover sufficient thread-level parallelism, tipically because of the necessarily conservative approach taken by the compiler. This normally results in the compiler including many unnecessary inter-thread communication/synchronization operations, or more likely, concluding that insufficient parallelism exists. Recent studies have proposed the use speculative thread-level parallelism (SpMT). This technique reduces the execution time of applications by executing several speculative threads in parallel.

These threads are speculative in the sense that they may be data and control dependent on previous threads and their correct execution and commitment is not guaranteed. Additional hardware/software is required to validate these threads and eventually commit them.

There are two main strategies for speculative TLP: 1) the use of helper threads to reduce the execution time of high-latency instructions by means of side effects [2][4][10][17][24], and 2) relaxing the parallelization constraints and parallelizing the applications into speculative threads ([1][3][11][18][19][21] among others).

The Helper Thread paradigm is based on the observation that the cost of some instructions severely impact performance, such as loads that miss in cache or branches that are incorrectly predicted. This paradigm attempts to reduce the execution time of the application by using speculative threads to reduce the cost of these high-latency operations.

In speculative parallelization, each of the speculative threads executes a different portion of the program. This partitioning process is based on relaxing the parallelization constraints and allowing the spawning of speculative threads even where the compiler cannot guarantee correct execution. Once the thread finishes, the speculative decisions are verified – if they were correct, then the application has been accelerated. If a misspeculation (control or data) has occurred, then the work done by the speculative thread is discarded and the processor continues with the correct threads.

The Mitosis processor is based on the speculative parallelization approach. The primary distinction from previous works stems from how inter-thread dependences are handled. It is possible to partition a program into enough parallel threads such that there are few or no dependences between them [19]. However, for most programs it is necessary to create threads where there are control/data dependences across these partitions to fully exploit the available parallelism. The manner in which such dependences are managed critically affects the performance of the SpMT processor [11]. Previous approaches have used hardware communication of produced register values [9], explicit synchronization [6][21], and value prediction [12] to manage these dependences.

In contrast, in the Mitosis framework we propose a new, software approach to manage both data and control dependences among threads. We find that this both produces values earlier than the hardware-assisted value-passing approaches, and more accurately than hardware value prediction approaches (due to the use of the actual application code). Each speculative thread is prepended with a speculative pre-computation slice (p-slice) that precomputes live-ins, while executing in a fraction of the time of the actual code that produces those live-ins.

In this paper, we present the Mitosis processor, a hardware/software approach to effectively exploit speculative TLP. The Mitosis framework is composed of a compiler that partitions the applications into speculative threads and a speculative multithreaded processor that is able to manage multiple speculative threads. It also provides novel mechanisms to track and manage the different versions of the memory and the register values for the speculative threads. The overall Mitosis processor framework is presented, including the compiler architecture, the hardware architecture, and several novel aspects of each. Some very promising early performance results are presented as well. This study focuses on applications that sophisticated parallelizing compilers cannot parallelize. Performance results reported by Mitosis processors show an average speed-up of about 2.2x for a subset of the Olden benchmark suite and 1.75x over a configuration that models perfect L1 level caches.

The rest of the paper is organized as follows: Section 2 presents the execution model. The different components of the Mitosis processors are described: the compiler in Section 3 and the microarchitecture in Section 4. Section 5 presents the performance evaluation of this architecture. Finally, Section 6 summarizes the main conclusions of the work.

2. Execution Model of the Mitosis Processor

Mitosis is a framework that combines hardware and software techniques to exploit speculative TLP. The Mitosis compiler is responsible for partitioning the application into speculative threads. It also plays an important role in dealing with data dependences since it generates the code to compute the live-in values of the speculative threads. The Mitosis processor architecture provides the hardware support for executing speculative threads and detecting any possible misspeculation.

Figure 1 shows the execution model of the Mitosis processor. Programs are partitioned into speculative threads statically with the Mitosis compiler. The partitioning process done by the compiler is described in Section 3. It basically explores the application to insert spawning pairs, that is, pairs of instructions made up of the point where the speculative thread will be spawned (the spawning point, or SP for short) and the point where the speculative thread will start its execution (the control quasi-independent point, or CQIP for short) [13]. The Mitosis compiler also computes the p-slice of each spawning pair that predicts the input values of the speculative thread.

This new binary obtained with the Mitosis compiler is executed on the Mitosis processor. Applications run on a Mitosis processor in the same way as on a conventional superscalar processor. However, when a spawn instruction is found, the processor looks for the availability of a free context (or thread unit). On finding a free one, the p-slice of the corresponding speculative thread is assigned to be executed at that thread unit. The p-slice ends with an unconditional jump to the CQIP, thereby starting the execution of the speculative thread. If no free thread unit is available when the spawn instruction is executed, the system looks for a speculative thread that is more speculative (further in sequential time) than the new thread we want to spawn. If any is found, the most speculative one is cancelled and its thread unit is assigned to the new thread.

Threads in Mitosis processors are committed in program order. The thread executing the oldest instructions in program order is non-speculative, whereas the rest are speculative. When any running thread reaches the CQIP of any other active thread, it stops fetching instructions until it becomes the non-speculative thread. Then, a verification process checks that the next speculative thread has been executed with the correct input values. If the speculation has been correct, the non-speculative thread is committed and its thread unit is freed. Moreover, the next speculative thread becomes the non-speculative. If there is a misspeculation, the next speculative thread and all its successors are squashed, and the non-speculative thread continues executing the instructions beyond the CQIP.

In Mitosis processors, the spawning process is highly general. Any speculative or non-speculative thread can spawn a new thread upon reaching an SP. Moreover, speculative threads can be spawned out of program order, that is, threads can be created in a different order than they will be committed.

3. Mitosis Compiler

The Mitosis compiler has been developed on top of the Open Research Compiler (ORC)[8]. The ORC infrastructure is a research IPF compiler that produces code with a quality similar to that of a production compiler. The different Mitosis modifications have been implemented in the back-end of the compiler after all the optimizations and before bundle formation. These modifications include the following highly coupled steps: 1) identification of spawning pairs, and 2) generation and optimization of the p-slice. We will describe these two steps below.

3.1. Spawning Pair Identification

The partitioning of applications into speculative threads is performed by means of the identification of spawning pairs. A spawning pair represents two instructions: the spawning point (SP) and

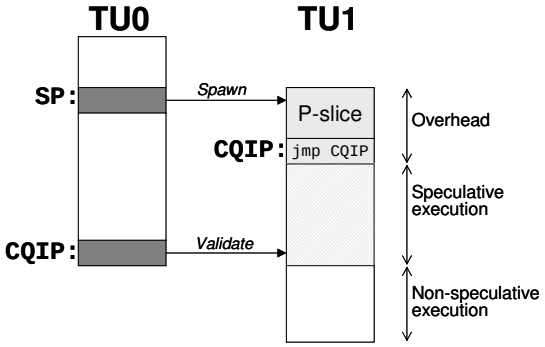


Figure 1. Mitosis execution model

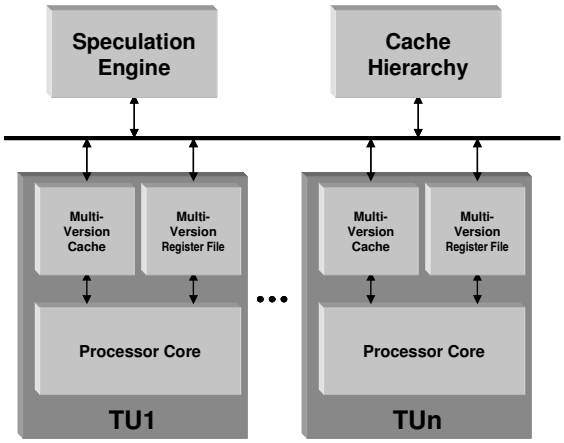


Figure 2. Mitosis processor μarchitecture

the control quasi-independent point (CQIP). This pair of instructions could be formed by any pair of instructions in the program, but there are some requirements they should meet in order to provide some benefit. Examples of these requirements are control and data independence, load balance issues, thread length, etc.

In this work, we propose a static mechanism that selects the spawning pairs by means of a cost model that estimates the expected benefit of any potential spawning pair. This model also considers the probability/cost of misspeculations and the interaction with other spawning pairs selected.

The Mitosis compiler uses edge profiling information, as extracted by many conventional compilers, to initially obtain a list of possible spawning pair candidates. This initial set of candidates is obtained pruning the whole set by a certain control independence and a minimum distance between the SP and the CQIP. These candidates have the SP and the CQIP in the same routine.

Once this step has been performed, the p-slice for each of these potential spawning pairs is calculated. The way slices are generated is described in the next subsection. Candidates are pruned again depending on the size of the p-slice. Large p-slices imply fewer overlapped instructions. Thus, those spawning pairs whose p-slice size, relative to the expected size of the body of the thread, is higher than a certain threshold are discarded.

Next, a synthetic trace of the program is generated from the edge profile information and a greedy algorithm is applied to get the best set of spawning pairs. In this algorithm, the pair that performs the best individually based on the cost model is selected among all the candidate spawning pairs . Taking into account this inserted thread, this process is repeated until the increasing benefit between two consecutive iterations is lower than a certain threshold.

3.2. Slice Generation

Given a spawning pair, a p-slice is the subset of the instructions executed between the spawning point and the control quasi-independent point needed for computing values used beyond the control quasi-independent point. Therefore, the first step to get the p-slice is to determine the thread live-in values. The compiler examines the code following the CQIP, but only for as many instructions as the average distance between the spawning point and the control quasi-independent point (because once the spawning thread reaches the CQIP and verifies the spawned thread, all values are available non-speculatively). Then, for these thread input values, it is determined which of them are produced

between the spawning pair and the corresponding data and control dependence graph for only those values is built. Those instructions that are in the sub-graphs of the thread live-ins are selected to be inserted in the slice. Finally, the slice ends with an unconditional branch to the CQIP.

The identification of the thread input values is relatively straightforward for register values, but harder for memory values. To detect thread input memory values, the Mitosis compiler uses an improved version of the memory dependence profiling.

Subroutine calls within the spawning pair that belong to the sub-graph of the thread input values are also included in the p-slice. However, subroutines that perform system calls are not included and then spawning pairs that require a system call in the p-slice are not considered for selection.

A key observation for generating p-slices is that they do not need to be correct. This is because the hardware, as described in the next section, will validate them and squash the thread when they are incorrect. Therefore, the compiler includes aggressive, sometimes unsafe, optimizations such as thread and slice branch pruning and memory dependence speculation. Complete information regarding the compiler support and the applied p-slice optimizations can be found at [14].

4. Mitosis Processors

The Mitosis processor has a multi-core design similar to an on-chip multiprocessor (CMP), as shown in Figure 2. Each thread unit executes a single thread at a time, and it is similar to a superscalar core. Each speculative thread may have a different version for each logical register and memory location. The different versions are supported by means of a local register file and a local memory per thread unit. In this section, the most relevant components of the microarchitecture are described, including the Multi-Version Register File and the Multi-Version Memory.

4.1. Spawning process

A speculative thread starts when any active thread fetches a spawn instruction in the code. The spawn instruction triggers the allocation of a thread unit, the initialization of some registers, and the ordering of the new thread with respect to the other active threads. These tasks are handled by the Speculation Engine.

To initialize the register state, the spawn instruction includes a mask that encodes which registers are p-slice live-ins. Those registers included in the mask are copied from the parent thread to the spawned one. On average, we have observed that just 6 registers need to be initialized for our benchmarks.

Any active thread is allowed to spawn a new thread when it executes a spawn instruction. Additionally, speculative threads can be spawned out of program order, that is, a speculative thread that would be executed later than another thread if executed sequentially can be spawned and executed in reverse order in a Mitosis processor. Some previous studies have shown that out-of-order spawning schemes have a much higher performance potential than in-order approaches [1] [11].

It is also necessary to properly order the spawned thread with respect to the rest of the active speculative threads. The order among threads will determine where a thread is going to look for values not produced by itself. Akkary and Driscoll [1] propose a simple mechanism in which the new spawned thread is always assumed to be the most speculative. On the other hand, Marcuello and González [11] propose an order predictor based on the previous executions of the speculative threads. In this work, this latter scheme is used since it provides better hit ratio (98% with the configuration described in Section 5).

4.2. Multi-Version Register File

To achieve correct execution and high performance, the architecture must simultaneously support the following, seemingly conflicting, goals: a unified view of all committed register state, the co-existence of multiple versions of each register, register dependences that cross thread units, and a latency similar to a single local register file.

This support is provided by the Mitosis multi-version register file, shown in Figure 3. As can be observed, the register file has a hierarchical organization. Each thread unit has its own Local Register File (LRF) and there is a Global Register File (GRF) for all the thread units. There is also a table, the Register Versioning Table (RVT) that has as many rows as logical registers and as many columns as thread units, that tracks which thread units have a copy of that logical register.

When a speculative thread is spawned in a thread unit, some register values are copied from the parent thread. These registers are encoded in the spawn instruction, as described above. Slices have the characteristic that they are not more nor less speculative than the parent thread; in fact, they execute a subset of instructions of the parent thread so the speculation degree is the same. Therefore, it needs to access the same register versions as the parent thread would see at the spawning point (while executing in a different thread unit).

When a thread requires a register value, the LRF is checked first. If the value is not present, then the RVT is accessed to determine which the closest predecessor thread that has a copy of the value. If there are no predecessors that have the requested register, then the value is obtained from the GRF.

There is an additional structure used for validation purposes: the Register Validation Store (RVS). When a speculative thread reads a register for the first time and this value has not been produced by the thread itself the value is copied into this structure. Additionally, those register values generated by the p-slice that have been used by the speculative thread are also inserted in the RVS. When this thread is validated, the values in the RVS are compared with the actual values of the corresponding registers in the predecessor thread. Doing so we ensure that values consumed by the speculative thread would have been the same in a sequential execution. Because we explicitly track values consumed, incorrect live-ins produced by the slice that are not consumed do not cause misspeculation.

Finally, when the non-speculative thread commits, all the modified registers in the LRF are copied into the GRF. An evaluation of the performance impact of the Multi-Version Register File design has been done. On average, more than 99% of the register accesses are satisfied from the LRF for the benchmarks evaluated. Thus, the average perceived latency for register access is essentially equal to the latency of the LRF, meeting the goals for our register file hierarchy.

4.3. Multi-Version Memory System

Similar to the register storage, the memory system provides support for multi-versioning, that is, it allows different threads to have different values for the same memory location. As shown in Figure 4, each thread unit has its own L0 and L1 data caches, which are responsible for maintaining the speculative values, since speculative threads are not allowed to modify main memory. Moreover, three additional structures are needed at each thread unit – the Slice Buffer (SB), the Old Buffer (OldB) and the Replication Cache (RC). Finally, there is a global L2 cache shared among all the thread units which can only be updated by the non-speculative thread, and a centralized logic to handle the order list of the different variables, the Version Control Logic (VCL).

The architecture of this memory system is inspired by the Speculative Versioning Cache (SVC) proposed by Gopal et al. [7] with notable extensions to handle p-slices. As a summary, the Mitosis memory subsystem contains the following novel features: support for p-slice execution and the replication cache. This section focuses on these new features.

A load in a p-slice needs to read the value of that memory location at the SP, while a load in the

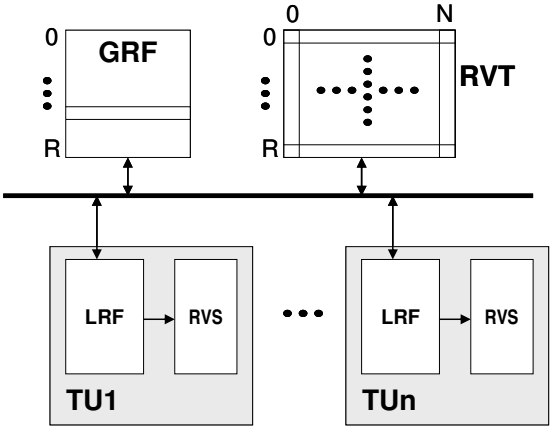


Figure 3. Multi-version register file

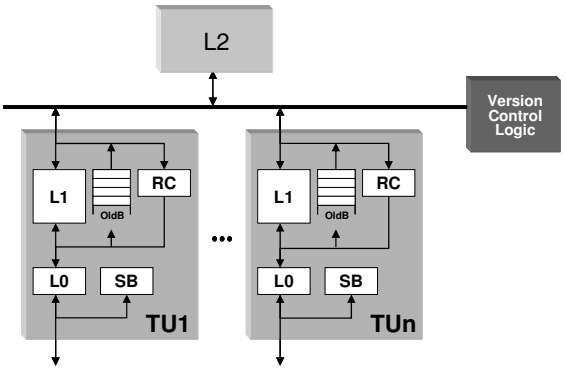


Figure 4. Multi-version memory system

thread needs to see the value at the CQIP, if it has not been produced by itself. For this reason, during the execution of a p-slice the processor needs to have an exact view of the machine state of the parent at the time of the spawn instruction. Therefore, when a thread performs a store and any of its children is still executing its p-slice, the value of that memory location needs to be saved before overwriting it since its children may later require that value. The buffers used for storing the values that are overwritten while a child is executing a p-slice are referred to as Old Buffers. Each thread unit has as many Old Buffers as direct child threads are allowed to be executing a p-slice simultaneously. Thus, when a speculative thread that is executing the p-slice performs a load to a memory location, it first checks for a local version at its local memory. In case of a miss, it checks in its corresponding Old Buffer from the parent thread. If the value is there, then it is forwarded to the speculative thread. Otherwise, it looks for it at any less speculative thread cache. When a speculative thread finishes its slice, it sends a notification to its parent thread to deallocate the corresponding Old Buffer. Finally, it is possible that a thread finishes its execution and some of its children are still executing the p-slice. Then, those Old Buffers cannot be freed until these threads finish their corresponding p-slices.

The values read during the execution of the p-slice have to be marked in some way to avoid being read by any other thread, and mistaken for state expected to be valid across CQIPs. To prevent a more speculative thread from reading an incorrect value, a new bit is added to the SVC protocol that is referred to as Old Bit. When a thread that is executing the slice performs a load from the parent Old Buffer, the value is inserted into the local cache with the Old Bit set. Then, when a more speculative thread requests this value, if it finds the Old Bit set it knows that the value stored in that cache may be potentially old and is ignored. Finally, when the slice finishes, all the lines of the local cache with the Old Bit set are invalidated.

Slice Buffers are used to store the values computed by the live-in p-slice in order to validate if the speculative thread is being executed with the correct input values. When a thread is allocated to a thread unit, the Slice Buffer is reset and all the stores performed during the execution of the slice go directly to Slice Buffer, bypassing the cache. Each entry of the Slice Buffer contains an address, a value, a read bit and a valid bit. Thus, when the slice finishes and the speculative thread starts its execution, every time a value is read from memory, the Slice Buffer is checked first. If the value is there, the read bit is set and the value copied to cache. When the thread becomes the non-speculative one, all the values that have been consumed from the Slice Buffer have to be checked for their

correctness. Thus, those entries that have their read bit set are sent to the previous non-speculative thread to validate their values.

When threads only exploit samethread memory locality, cache miss rates would be extreme. Preliminary experiments showed that the miss ratio of the L1 cache was very high for the speculative threads. This is due to the fact that when a thread commits, all the lines of the local cache set its commit bit to defer the traffic burst to main memory. As a result, every newly spawned thread begins with a completely empty cold cache. The impact of this feature could be reduced by introducing to the SVC protocol the stale bit as was proposed elsewhere [7]. However, this memory model has the problem of very poor locality exploitation. If all the active threads consume the same memory line, values have to travel from one thread unit to another every time. Mitosis solves that with the Replication Cache (RC). This small cache works as follows: when a thread performs a store, it has to send a bus request to know if any more speculative thread has performed a load on that address. We use this request to send the value and store it in the Replication Cache of all the threads that are more speculative as well as in the free thread units. Thus, when a thread performs a load, L1 and the Replication Cache are checked simultaneously. If the value requested is not in L1 but it is in the Replication Cache, the value is moved to L1 and supplied to the thread unit. This simple mechanism prevents the thread units from starting with cold caches as well as taking advantage of locality.

4.4. Thread Validation

A thread finishes its execution when it reaches the starting point of any other active thread, that is, the CQIP. At this point, if the thread is nonspeculative, it validates the next thread. Otherwise, it waits until it becomes the non-speculative one. The first thing to verify is the order. The CQIP found by the terminating thread is compared with the control quasi-independent point of the following thread in the thread order (as maintained by the order predictor). If they are not the same, then an order misspeculation has occurred and the following thread and all its successors are squashed.

If the order is correct, then the thread input values used by the speculative thread are verified. These comparisons may take some time, depending on the number of values to validate. We have observed that on average, for our workloads, a thread validation requires to check less than 1 memory and about 5 register values for this validation. Note that only memory values produced by the slice and then consumed by the thread (values read from the slice buffer) need to be validated when the previous thread finishes. Other memory values consumed by the thread are dynamically validated as soon as they are produced through the versioning protocol described above. If no misspeculations are detected, the non-speculative thread is committed and the next thread becomes the nonspeculative one. The thread unit assigned to the finished thread is freed, except when there is a child thread that is still executing the p-slice since it may require values available in the Old Buffer.

5. Experimental Framework

The performance of the Mitosis processors was evaluated through a detailed execution-driven simulation. The Mitosis compiler has been implemented on top of the ORC compiler to generate IPF code. The Mitosis processor simulator models a research Itanium CMP processor with 4 hardware contexts based upon SMTSIM [23]. The main parameters considered are shown in Table 1. The numbers in the table are per thread unit.

To evaluate the potential performance of the Mitosis architecture, a subset of the Olden suite [16] has been used. The benchmarks used are the bh, em3d, health, mst and perimeter, with an input set that on average executes around 300M instructions. Statistics in the next section correspond to the whole execution of the programs. Different input data sets have been used for profiling and

Mitosis processor configuration

Fetch, in-order issue and commit bandwidth	2 bundles (6 instructions)	**Order Predictor size**	16K-entry
Reorder buffer	512 instructions	**Branch Predictor**	Local Gshare
I-Cache	64KB	**Local Register File**	1 cycle
L0-Cache	4-way 16 KB – hit: 1 cycle	**Global Register File**	256-entry / 6 cycles
L1-Cache	4-way 1 MB – hit: 3 cycles	**Spawn overhead**	5 cycles
Crossfeed latency	3 cycles	**Validation overhead**	15 cycles
Replication cache	4-way 16 KB	**Slice buffer**	1K-entry – hit: 1 cycle
L2-Cache (shared)	4-way 8 MB – hit: 6 cycles; miss: 250 cycles	**Old buffer**	3 – 128-entry each

simulation. The rest of the suite has not been considered due to the recursive nature of the programs. Up to this point, the Mitosis compiler is not able to extract speculative TLP in recursive routines. This feature will be targeted in future work.

Olden benchmarks have been chosen since they are pointer intensive programs and automatic parallel compilers are unable to extract TLP. To corroborate this, we have compiled the Olden suite with the Intel C++ production compiler which produces parallel code. Almost none of the code was parallelized by this compiler.

5.1. Performance Figures

Table 2
Characterization of the Olden benchmarks

Benchmarks	#Spawned threads	Thread size	Slice size	%Slice / thread	Thread live-ins	%Squashes
bh	422	15543.0	196.5	1.3	4.4	0.7
em3d	396638	422.1	9.0	2.1	1.0	0.3
health	198497	1112.7	41.6	3.7	2.7	26.9
mst	1367114	271.4	5.8	2.1	2.3	0.8
perimeter	493725	576.8	24.0	4.2	3.6	1.0
AMEAN	**491279.2**	**3585.2**	**55.4**	**2.7**	**2.8**	**6.0**

Statistics corresponding to the characterization of the speculative threads are shown in Table 2. The last row shows the arithmetic mean for the evaluated benchmarks. The first column shows the number of spawned threads by benchmark and the second column the average number of speculative instructions executed by the speculative threads. It can be observed that bh spawns the fewest number of threads but their average size is about 30 time larger than for the rest of benchmarks. On the other hand, mst spawns the most but their average size is the lowest. The third column shows the average dynamic size of the slices and the fourth column the relationship between the sizes of the speculative threads and their corresponding slice. This percentage is consistently quite low for all the studied benchmarks and on average represents less than 3%. The fifth column shows the average number of thread input values that are computed by the slice, that is, on average it is only necessary to compute 3 values to execute the speculative threads. Finally, the right-most column represents the average number of squashed threads. For all the benchmarks, this percentage is rather low except for health where almost 1 of every 4 threads is squashed. We have observed that for this particular benchmark,

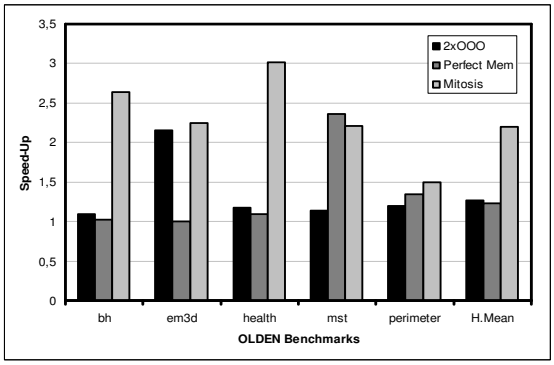


Figure 5. Speed-up over single thread

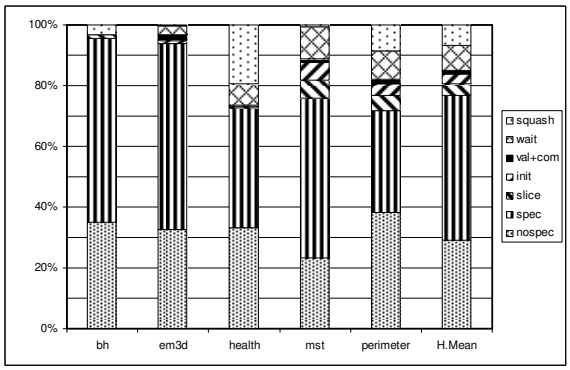


Figure 6. Time breakdown for Mitosis

memory dependences for the profiling and simulated inputs are significantly different, which result in many memory dependence misspeculations.

Figure 5 shows the speedups of the Mitosis processor over a superscalar in-order processor with about the same resources as one Mitosis thread unit with no speculative threading. For comparison, we also show the speedup of a more aggressive processor, with twice the amount of resources (functional units), twice the superscalar width and out-of-order issue (with no speculative threading), and a processor with perfect first-level cache (an aggressive upper limit to the performance of helper threads that target cache misses).

It can be observed that the Mitosis processor achieves an average speed-up close to 2.2x over single-threaded execution, whereas the rest of the configurations provide much lower performance. Perfect memory achieves a speed-up of just 1.23x and the more aggressive out-of-order processor only provides a 1.26x speed-up. Some results worth further discussion are the similar performance achieved by the Mitosis processor compared with the outof- order double-wide core in em3d and compared with the perfect memory model in mst. In the former case, ILP is quite abundant in this benchmark, which could be additionally exploited with more complex Mitosis configurations (with out-of-order thread units). In mst, the performance of the memory system is quite low (for a single-threaded execution, the miss ratio in L1 cache is higher than 70% and close to 50% in L0). Even so, Mitosis is surprisingly competitive with the perfect memory configuration (within 8%). In summary, we find Mitosis mirrors an aggressive superscalar when ILP is high, an unattainable memory subsystem when memory parallelism is high, and outperforms both when neither ILP nor memory parallelism is high.

Figure 6 shows the time breakdown for the execution of the different benchmarks in the Mitosis processors. As expected, most of the time the thread units are executing useful work (the sum of the non-speculative and the speculative execution). On average, this percentage is almost the 80% of the time the thread units are working and higher than 90% for bh and em3d. The overhead added by this execution model represents less than 20% for these benchmarks. The most significant part of this overhead comes from the wait time. This time stands for the time a thread unit has finished the execution of a speculative thread but it has to wait until becoming non-speculative to commit. The other components of the overhead are the slice execution, the initialization overhead, and the validation and commit overhead. It is worth noting that the execution of the slices only corresponds to 4%. Finally, the top of the bars shows the average time thread units are executing incorrect work, that is, threads that are squashed. This percentage is only 8% overall, mostly due to health, where

the overhead is almost 20%. In this case, most of the squashes are due to memory violations and the cascading effect of the squashing mechanism. Recall, however, that health still maintains a 3x speedup despite these squashes.

These results strongly validate the effectiveness of the execution model introduced in this paper (precomputation slice based speculative multithreading) in meeting the Mitosis design goals: high performance, resulting from high parallelism (low wait time), high spawn accuracy (very low squash rates), and low spawn and prediction overhead (very low slice overheads).

6. Conclusions

In this work, we have presented and evaluated the Mitosis framework, which exploits speculative TLP through a novel scheme to deal with interthread data dependences. This model is based on predicting via software the thread live-ins. It does so by means of inserting a piece of code in the binary that speculatively computes the values at the starting point of the speculative thread. This code, referred to as a p-slice, is built from a subset of the code after the spawn instruction and before the beginning of the speculative thread. A key feature of Mitosis processors is that p-slices do not need to be correct, which allows the compiler to use aggressive optimizations when generating them.

An efficient mechanism to partition the code into speculative threads is presented. This mechanism looks into the code to detect which parts of the program will provide the highest benefit, taking into account possible misspeculations, overheads, and load balancing. Moreover, some compiler optimization techniques have been presented in order to reduce the weight of the slices in the speculative threads.

The key microarchitecture components of Mitosis processors have been presented: (1) hardware support for the spawning, execution, and validation of p-slices allows the compiler to create slices with minimal overhead, (2) a novel multi-version register file organization supports a unified global register view, multiple versions of register values, transparent communication of register dependences across processor cores, all with no significant latency increase over traditional register files, and (3) a memory system that support multiple versions of memory values, and introduces a novel architecture that pushes values towards threads that are executing future code, to effectively mimic the temporal locality available to a single-threaded processor with a single cache.

Finally, the results obtained by the Mitosis processor with 4 thread units for a subset of the Olden benchmarks are impressive. It outperforms the single-threaded execution by 2.5x, and more than 1.75x compared with a double-size out-of-order processor, and over a perfect memory model. These results confirm that there are large amounts of TLP on codes that can be extracted by speculative techniques such as those of Mitosis on codes that cannot be parallelized by conventional approaches.

Acknowlegdments

We would like to thank Peter Rundberg (currently at Gridcore, Sweden), Hong Wang and John Shen (at Intel Labs, Santa Clara) and Dean Tullsen (at UC San Diego) for his valuable collaboration in this work. Also, we would like to thank the ORC team for their support in the compiler implementation. This work has been partially supported by the Spanish Ministry of Education and Science under contract TIN2004-03072 and Feder funds.

References

[1] H. Akkary and M.A. Driscoll: A Dynamic Multithreading Processor. Procs. of the 31st Int. Symp. on Microarchitecture. 1998

[2] R.S. Chappel et al.: Simultaneous Subordinate Microthreading (SSMT). Procs. of the 27th Int. Symp. on Computer Architecture. 2000

[3] L. Codrescu and D. Wills: On Dynamic Speculative Thread Partitioning and the MEM-Slicing Algorithm. Procs. of the Int. Conf. on Parallel Architectures and Compilation Techniques. 1999

[4] J.D. Collins et al: Speculative Precomputation: Long Range Prefetching of Delinquent Loads. Procs. of the 28th Int. Symp. on Computer Architecture. 2001

[5] K. Diekendorff: Compaq Chooses SMT for Alpha. Microprocessor Report(December). 1999

[6] M. Franklin and G.S. Sohi: The Expandable Split Window Paradigm for Exploiting Fine Grain Parallelism. Procs. of the 19th Int. Symp. on Computer Architecture, pp. 58-67, 1992.

[7] S. Gopal, T.N. Vijaykumar, J.E. Smith and G.S. Sohi: Speculative Versioning Cache. Procs. of the 4th Int. Symp. on High Performance Computer Architecture. 1998

[8] http://ipf-orc/sourceforge.net

[9] V. Krishnan and J. Torrellas: Hardware and Software Support for Speculative Execution of Sequential binaries on a Chip-Multiprocessor. Int. Conf. on Supercomputing. 1998

[10] C. Luk: Toleraing Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors. Procs. of the 28th Int. Symp. on Computer Architecture. 2001

[11] P. Marcuello: Speculative Multithreaded Processors. Ph.D. Thesis, Universitat Politecnica de Catalunya. 2003

[12] P. Marcuello, J. Tubella and A. González: Value Prediction for Speculative Multithreaded Architectures. Procs. of the 32nd. Int. Conf,. on Microarchitecture. 1999

[13] P. Marcuello and A. González: Thread-Spawning Schemes for Speculative Multithreaded Architectures. Procs. of the 8th Int. Symp, on High Performance Computer Architectures. 2002

[14] C. García et. al. : Mitosis Compiler: an Infrastructure for Speculative Threading Based on Pre-Computation Slices. Proc. of the Int. Symp. on Programming Language Design and Implementation. 2005

[15] T. Marr et al.: Hyper-threading Technology Architecture and Microarchitecture. Intel technology Journal, 6(1). 2002

[16] A. Rogers, M. Carliste, J. Reppy and L. Hendren: Supporting Dynamic Data Structures on Distributed Memory Machines. ACM Trans on Programming Languages and System (March). 1995

[17] A. Roth and G.S. Sohi: Speculative Data-Driven Multithreading. Procs. of the 7th. Int. Symp. on High Performance Computer Architecture. 2001

[18] G.S. Sohi, S.E. Breach and T.N. Vijaykumar: Multiscalar Processors. Procs. of the 22nd Int. Symp. on Computer Architecture. 1995

[19] J. Steffan and T. Mowry: The Potential of Using Thread-level Data Speculation to Facilitate Automatic Parallelization. Procs. of the 4th Int. Symp. on High Performance Computer Architecture. 1998

[20] S. Storino an dJ. Borkenhagen: A Multithreaded 64-bit PowerPC Commercial RISC Processor Design. Procs. Of the 11th Int. Conf. on High Performance Chips. 1999

[21] J.Y. Tsai and P-C. Yew: The Superthreaded Architecture: Thread Pipelining with Run-Time Data Dependence Checking and Control Speculation. Procs. of the Int. Conf. on Parallel Architectures and Compilation Techniques. 1995

[22] M. Tremblay et al.: The MAJC Architecture, a synthesis of Parallelism and Scalability. IEEE Micro, 20(6). 2000

[23] D. M. Tullsen, S.J. Eggers and H.M. Levy: Simultaneous Multithreading: Maximizing On-Chip Parallelism. Procs. of the 22nd Int. Symp. on Computer Architecture. 1995

[24] C.B. Zilles and G.S. Sohi: Execution-Based Prediction Using Speculative Slices. Procs. of the 28th Int. Symp. on Computer Architecture. 2001

Grid Computing

A Paradigm for Allocating Parallel Application Tasks to Heterogeneous Computing Resources on the Grid

Bassel Arafeh, Khaled Day, and Abderezak Touzene
Department of Computer Science
Sultan Qaboos University
Muscat, Oman

Abstract

The work addresses the problem of allocating parallel application tasks for execution on heterogeneous computing resources on the Grid. The proposed allocation paradigm considers issues pertinent to the Grid environment. Basically, our model considers the relationship between the clients and the environment in one side, and the relationship between the system providers and the environment on the other. This consideration is reflected in utilizing the client and the system specification to determine the objective function and the constraints of the mapping problem. The paradigm adopts a multilevel graph partitioning and mapping approach. The objective of the mapping is to minimize the parallel application execution time, subject to the specified constraints. The paradigm introduces an efficient heuristic for the coarsening step, called the VHEM method. The simulation study shows that the heuristic can achieve very high reduction factor, when the ratio of the number of tasks to the number of processors exceeds a threshold value. Also, the paradigm introduces an efficient heuristic for the refinement phase, in which, the space of processor preference for remapping includes the subset of processors on the shortest paths from the currently allocated processor to all other processors to which adjacent vertices are allocated.

1. Introduction

The concept of clustering computing resources to solve computational problems has been the focus of high-performance computing community for more than two decades. The advances in high-speed microprocessors and computer networks have made cost-effective parallel computing based on clusters or networks of workstations (NOWs) an alternative to expensive supercomputers. However, the demand for computing power continues to grow, while most of the available machines are eventually underutilized. Recently, there has been a proposal for using large-scale high-performance distributed computing resources through a new architecture, known as the computational Grid [4].

In order to construct a Grid computing environment, it is very important to have a Grid Resource Management System (RMS). The basic functions of an RMS would be to accept requests for resources by users' applications and allocate computing resources to those requests from the overall pool of the grid resources. The RMS is configured as a middleware infrastructure software system, through which resource information is disseminated, suitable resources are discovered, and applications are mapped and scheduled for execution.

This work builds on the concept of Grid application schedulers, which has been adopted in research projects such as the AppLes and GrADS projects [1] [3]. Our work focuses on the allocation of tasks of a resource-intensive parallel application to a selected pool of Grid resources for execution. The objective is to find a matching between the tasks and the set of Grid computing resources that optimizes the application completion time. We assume a static and decentralized approach, where a Grid application scheduler works on a predictive estimation of the application resource requirements provided by the client, such as the reservation period and the execution behavior. On the other

side, it works on a predictive estimation of the characteristics of the selected Grid resources for the application, such as the duration of a resource-sharing period, the resource utilization factor, and the CPU speed. In general, it has been shown by Bokhari[2] that the general mapping problem is NP-hard. Several heuristic methods have been proposed to provide approximate solutions for parallel architectures. However, the mapping problem in computational Grids has received little attention so far. In our current work, we address the mapping problem in the context of a computational Grid based on the multilevel graph partitioning scheme [5] [6] [7]. The rest of the paper is organized as follows. Section 2 introduces our application model, system model, assumptions and the problem statement. Section 3 describes our proposed heuristic for mapping tasks of a parallel application to a pool of Grid resources. Section 4 describes the simulation and performance evaluation results. Finally, section 5 is the conclusion.

2. Definitions and Background

2.1. System Model

The target architecture for the execution of a parallel application is a heterogeneous multi-cluster environment, formed from the distributed heterogeneous computing resources on the Grid. Accordingly, a heterogeneous computational system is modelled as a weighted undirected graph, $S = (P, L, \tau, \delta)$, referred to as the *system graph*. Where, P is a finite set of vertices representing sites/processors of the system on the Grid; and L is a finite set of edges representing the communication links between sites/processors. Each site/processor vertex, p, is characterized by a specified or announced processing weight, $\tau(p)$, reflecting its processing cost per unit of computation. Each edge, $l_{ij} = (p_i, p_j)$, has a link weight, $\delta(p_i, p_j)$, that denotes its communication latency (cost) per unit of communication between p_i and p_j. We assume each processor, $p \in P$, is characterized by a set of system parameters, based on its available resources for the Grid environment, (e.g., memory capacity, cpu speed, workload, operating system, etc.). For the purpose of simplifying the system model, we assume each processor, p, has a declared utilization factor, $\mu(p)$, denoting the local workload on the processor; and, a specified maximum duration, $\psi(p)$, for allowing its computational resources to be shared on the Grid environment. We will refer to $\psi(p)$ by the *processor's sharing period*.

The Grid does not enforce constraints on the network topolgy and the communication latency between processors. Therefore, an arbitrary network topology of the system is assumed. However, we assume the system graph is connected. Although the topology of the network is not completely connected, we can derive a communication latency matrix, $CL = [lat(p_i, p_j)]$, that represents the communication latency between any two processors in the network. The communication latency, $lat(p_i, p_j)$, between any two adjacent processors is equal to $\delta(p_i, p_j)$. While the communication latency, $lat(p_i, p_j)$, between any two non-adjacent processors is the sum of the link weights on the shortest path between them. The matrix is symmetric, since all links are assumed to represent full duplex communication.

2.2. Application Model

In this work, a weighted undirected graph $G = (V, E, \omega, \lambda)$, known as a task interaction graph (TIG), is used to model a parallel application, which we refer to as the *application graph*. Where, V is a finite set of vertices representing the application tasks; and E is a finite set of edges, $E = \{(v_i, v_j) |\ v_i, v_j \in V\}$, representing data dependency between two vertices, v_i and v_j. However, an edge (v_i, v_j) does not impose any precedence relation between the incident vertices v_i and v_j. Each vertex v has a computation weight $\omega(v)$ that represents the amount of computation required by this

task to accomplish a unit progress. Each edge $e_{ij} = (v_i, v_j)$ has a communication cost $\lambda(e_{ij})$ that represents the amount of data to be communicated between v_i and v_j to advance a unit progress. The execution behavior of the parallel application is assumed to pass through a number of iterations. Each iteration forms a unit progress in the execution behavior of the application, which consists of a communication phase followed by a computation phase. Therefore, we assume that the modelled application has a requirement for executing the tasks iteratively at a certain rate, ρ times per second, referred to as the *application execution rate*. Also, the application specifies a maximum duration Γ, referred to as the *application reservation period,* that reflects the total time to be reserved by the application on the computational Grid, in order to perform all required iterations.

2.3. Problem Definition

Given an application graph $G = (V, E, \omega, \lambda)$ and a system graph $S = (P, L, \tau, \delta)$, we need to find a mapping, $\Pi : V \longrightarrow P$, such that each vertex $v \in V$ is assigned to a partition $\pi(p)$ that is allocated to a processor, p, in the system graph for execution. The objective of the mapping is to minimize the application execution time, subject to the application requirements and the system constraints. In this work, we assume there is no overlapping between computation and communication. Therefore, the execution time of a task is determined by the summation of its computation time and all the communication costs with its dependent tasks. Accordingly, the execution time of a task v_i on a processor p is defined as

$$ET(v_i, p) = \omega(v_i) \times \tau(p) + \sum_{q \in P \ \& \ q \neq p} \sum_{v_k \in \pi(q)} \lambda(v_i, v_k) \cdot lat(p, q) \tag{1}$$

Where, $\pi(q)$ is a partition of the set of vertices, V, that is mapped to processor q. Then the execution time of partiton $\pi(p)$ on processor p is defined as

$$ET(\pi(p)) = \sum_{v_i \in \pi(p)} \{\omega(v_i) \times \tau(p) + \sum_{q \in P \ \& \ q \neq p} \sum_{v_k \in \pi(q)} \lambda(v_i, v_k) \cdot lat(p, q)\} \tag{2}$$

Hence, the parallel application execution time, ET, is given by

$$ET = \max_{p \in P} \{ET(\pi(p))\} \tag{3}$$

Then, the objective function for partitioning and mapping a parallel application to a heterogeneous system on the Grid is to minimize ET subject to the following

$$\Gamma \leq \psi(p)(1 - \mu(p)), \ \forall p \in P \tag{4}$$

$$ET(\pi(p)) \leq \frac{\psi(p)(1 - \mu(p))}{\rho \cdot \Gamma}, \ \forall p \in P \tag{5}$$

Where, $\rho \cdot \Gamma$ represents the maximum number of iterations that can be executed within a reservation period Γ for the parallel application. The inequality 4, specifies the relationship between the application reservation period, Γ, and the processors' sharing period, $\psi(p)$, and the utilization factor, $\mu(p)$. Inequality 4 indicates that the total time available on any processor, p, should not be less than the application's reservation period, Γ. This constraint may work as a rule for selecting a processor for the execution of the parallel application on the Grid environment. The system constraint in inequality 5 defines the amount of workload that is acceptable by each processor p, given it has a utilization factor $\mu(p)$, and a processor's sharing period $\psi(p)$. The amount of workload is determined based on the expected number of iterations to be performed by the application during a maximum reservation period Γ. Accordingly, the execution time of a partition, $\pi(p)$, should not exceed the amount of time that a processor, p, can allocate per iteration.

3. Multilevel Clustering, Mapping and Refinement Paradigm

In this section, we introduce a multilevel paradigm for clustering irregular TIG into a contracted graph with a reduced number of vertices. The process of contraction, referred to by coarsening, is carried out at several levels, until a threshold number of vertices is reached. The vertices of the coarsest TIG (CTIG) can be mapped to the system processors, with the objective of minimizing the application execution time. Assigning the vertices of the CTIG to the processors generates the initial graph partitions. However, the optimal or suboptimal initial partitioning and mapping may not be so for the original graph. An iterative optimization procedure can be applied at each level, through which a coarse graph is expanded or returned to its parent graph for further refinement. At each level of expansion, the optimization scheme is used to reduce the application execution time. The refinement approach is based on considering the possible migration of vertices to other processors, such that the maximum execution time among all processors is minimized. In the following, we introduce each phase of the paradigm as applied to mapping parallel applications to heterogeneous processors in the Grid environment.

3.1. Multilevel Clustering Phase

In this phase, the original graph (TIG) is contracted/coarsened into a sequence of smaller graphs, $G_i = (V_i, E_i, \omega_i, \lambda_i)$, starting from the original graph $G_0 = (V_0, E_0, \omega_0, \lambda_0)$, such that $|V_i| < |V_{i-1}|$. A coarser graph at level i is obtained from collapsing edges at level $i-1$. The collapse of an edge $e_{i-1}(v_1, v_2)$ at level $i-1$ generates a single vertex $u \in V_i$ at level i, where $\omega_i(u) = \omega_{i-1}(v_1) + \omega_{i-1}(v_2)$. The approach relies on finding a maximal independent subset of graph edges, or a matching of vertices, then collapsing them. Two edges are called independent if they are not incident on the same vertex. It follows that a subset of graph edges is independent and maximal if no more edges can be added to the subset without making two edges incident on the same vertex. The maximal independent subset of graph edges can be generated by visiting vertices in a random order, matching each unmatched vertex with one of its unmatched neighbors randomly.

Since the objective of the partitioning and mapping is to minimize the maximum execution time, it would be beneficial if the clustering phase can minimize the total communication cost in the CTIG. In sequal, the coarsening steps should collapse the most heavly weighted edges as proposed by Karypis and Kumar [5], referred to as heavy edge matching (HEM). In this work, we adopt a modified approach of the HEM. Only edges with communication costs exceeding the average edge communication cost are selected for matching a vertex with one of its unmatched neighbors. We call the scheme a Very Heavy Edge Matching (VHEM). As such, the VHEM guides the coarsening step towards achieving an effective reduction in the total communication cost at each level; while maintaining the ability for performing refinement at different resolution levels. But, the subset of collapsed edges is not maximal as defined previously. To overcome the drawback of this policy, we relax the issue of independence among the subset of collapsed edges. If an unmatched vertex has no unmatched neighboring vertices that are joined with a very heavy-edge weight, it is allowed to be matched with a matched neighbor, given that the cost of the edge joining them is also a very heavy-edge weight.

Furthermore, the coarsening phase must resolve three main issues. These are related to controlling the rate at which the size of a TIG is reduced, the threshold value for terminating the clustering phase, and the weight of a coarser vertex. For the first issue, the coarsening step at each level can be stopped when the size of the generated coarser graph becomes a certain factor (e.g., 1.5-2.0) of the finer graph. The aim is to control the rate at which a graph is reduced at each level, in order to allow refinement to take place at different resolution levels. For the second issue, the coarsening process

can be stopped when the number of vertices in the coarser graph becomes less than or equal to the number of processors in the system graph; and this is the approach taken in this work. For the third issue, the weight of a coarser vertex is constrained to be less than or equal to an upper-bound. The aim is to limit the rate at which the weight of certain vertices grow. The upper bound, $ub(v)$, of the weight of a coarser vertex v is determined from the the rate of iterative execution, ρ, the average communication latency in the system, Δ, the total communication cost, c_v^i, incident on a vertex $v \in V_i$ at level i, and the average processor speed, s_a. That is, $ub(v) = T(v) \cdot s_a$, where, $T(v)$ is the estimated maximum period of computation per iteration, and it is given by $T(v) = 1/\rho - \Delta \cdot c_v^i$.

3.2. Initial Mapping Phase

The second phase performs an initial mapping of the coarsest graph, $G_c = (V_c, E_c, \omega_c, \lambda_c)$, to the system processors, with the objective of minimizing the application's computation time. There is no need to apply a general optimization technique at this phase, since the refinement phase will apply an incremental refinement procedure at each uncoarsening step. Therefore, the initial mapping allocates tasks with heavy computation weights, $\omega(v)$, to processors having higher speeds. The application graph vertices, V_c, are sorted in descending order based on their computation weights. Similarly, the system graph processors are sorted in descending order based on their speeds. However, a processor's speed is adjusted to take into consideration the specified processor's workload. That is, the adjusted processor's speed is $s\prime(p) = s(p)(1 - u(p))$, where $s(p) = 1/\tau(p)$. Accordingly, the application's tasks are mapped to the system processors that have the same corresponding order.

3.3. Refinement Phase: A Greedy Remapping Approach for a k-way Partitioning

In this phase, the mapping, Π_c, of the coarsest graph G_c is projected back to the original graph G_o through several levels of refinements. Starting from Π_c, a mapping at level i, Π_i, is obtained from the mapping at level $i + 1$, Π_{i+1}, by assigning the mapping $\Pi_{i+1}(v)$ of each vertex $v \in V_{i+1}$ in the coarser graph G_{i+1} to each vertex $u \in V_i$ in the finer graph G_i, that has been merged to produce the vertex v. The operation is called the uncoarsening step, and the whole process is referred to by the uncoarsening phase. Associated with each uncoarsening step at level i there is a refinement step, which is applied to reduce the application execution time by checking for vertex migration across the boundaries of the partitions.

3.3.1. The Gain Function

Let p_m be the processor having the maximum execution time over all other processors due to the current mapping Π of G. Also, let $\Pi^{'}$ be the mapping of G if a vertex $v \in V$ migrates from a processor p_m to any other processor $p_q \in P$. Given a cost function $\Phi(\Pi)$ for a mapping Π, the fruitfulness of migrating a vertex $v \in \pi(p_m)$ to a processor p_q is found by a gain function $gain(v, p_m, p_q)$, such that

$$gain(v, p_m, p_q) = \Phi(\Pi) - \Phi(\Pi^{'}) \tag{6}$$

where,

$$\Phi(\Pi) = ET(\pi(p_m)) = \max\{ET(\pi(p))\} \text{ and } \Phi(\Pi') = ET(\pi(p_q)) + ET(v, p_q) \tag{7}$$

Hence, the gain function says that, in order to have a fruitful migration step for a vertex v from processor p_m to processor p_q, it must have a positive value greater than zero.

3.3.2. The Refinement Step

The gain function, $gain(v, p_m, p_q)$, for a vertex $v \in \pi(p_m)$ can be calculated with respect to every other partition $\pi(p_q)$, where $p_m \neq p_q$. Basically, the priority will be given to move a vertex v to a partition $\pi(p_q)$, if it produces the maximum gain in the cost function over all possible migrations of vertices on the boundary of $\pi(p_m)$ with other partitions.

Based on the cost function, the processor, p_m, having the maximum execution time, ET_{max}, must be considered the focal point for the next refinement step. The candidate vertices for migration are those on the borders of $\pi(p_m)$ with other partitions. A candidate border vertex v can be moved to a partition $\pi(p_q)$, only if the move of the vertex would not voilate the system constraint, that is

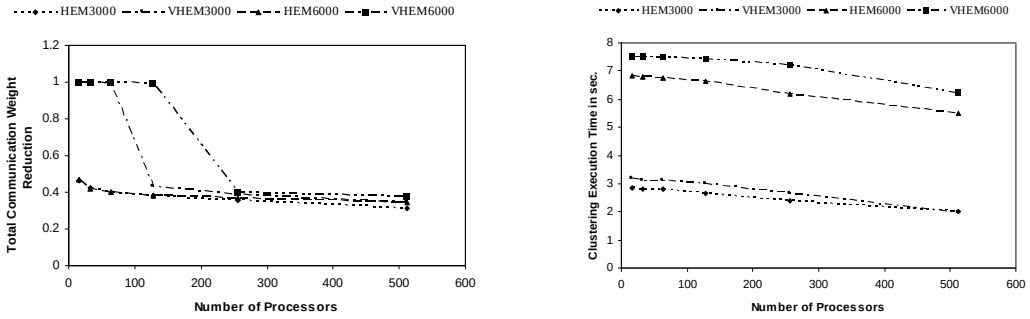
$$ET(\pi(p_q)) + ET(v, p_q) \leq \frac{\psi(p)(1 - \mu(p))}{\rho.\Gamma} \tag{8}$$

Our paradigm deviates from the general multilevel approach for partitioning in the way a processor preference is selected. In the general multilevel approach, a full processor preference may be considered, where a vertex may migrate to any other processor [6] [7]. In this case, the cost of a full processor preference is $O(|P|^2)$, since the computation of the gain function is $O(|P|\,.d_{av})$, and there are $(|P| - 1)$ possible migrations. Where, d_{av} is the average degree in the system graph. Other alternatives can be employed to reduce the cost by restricting the migration of vertices to adjacent partitions or to adjacent processors [7]. As such, the cost of computing the gain function for selecting the best target processor is unlikely to be $O(|P|^2)$. However, the later two methods may not guarantee to find the maximum gain. Therefore, we extend the possible space for vertex migrations to adjacent partitions/processors to include the subset of processors on the shortest paths from p_m to all neighboring processors for a border vertex. The approach is more costly than considering migrations to just adjacent partitions or adjacent processors. However, it is not expected to reach $O(|P|^2)$, if the number of processors is not too small.

4. Simulation and Performance Study

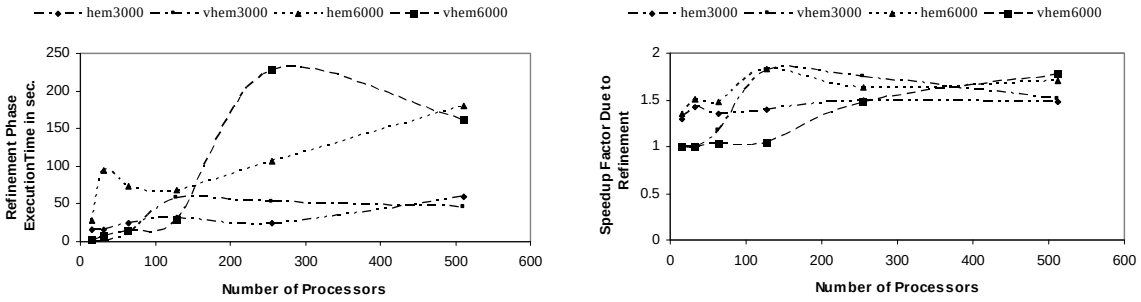
A simulation study of our paradigm for allocating application tasks to heterogeneous computing resources on the Grid has been conducted. All phases of the paradigm have been implemented and tested using randomly generated graphs for the application and system models. We have generated application graphs randomly with sizes ranging from 100 to 6000 vertices, based on a vertex degree in the range 2-20. For the system model, we have generated graphs representing arbitrary networks consisting of 16, 32, 64, 128, 256, and 512 processors. The system graphs are connected, with a maximum node degree equal to 0.25 of the graph size for all models; except for the 512 processors model, it is limited to a maximum of 32.

We have implemented the multilevel clustering phase based on both Karypis and Kumar HEM method and our VHEM method. The reduction factor goal for each coarsening step is set to be about 1.82. However, a coarsening step may stop as soon as the coarser graph size reaches a value less than or equal to the number of processors, or an upper bound of iterations has been reached. The performance evaluation of both methods for application graphs with 3000 and 6000 vertices for mapping to systems with different number of processors are shown in Figure 1. The plots in Figure 1(a) show that the VHEM method always generates coarsest graphs with less total communication volume than the HEM method. However, the cost of the VHEM method in terms of the execution time is slightly higher than the cost of the HEM method, as shown in Figure 1(b). But the major advantage of the VHEM method is in its ability to achieve a very high reduction in the total communication volume, when the ratio of the application graph size to the number of the processors is



(a) (b)

Figure 1. The clustering phase execution time and the total communication weight reduction versus the number of processors.



(a) (b)

Figure 2. Comparisons between the HEM and VHEM coarsening techniques on the speedup factor and the execution time of the refinement phase for application graphs with 3000 and 6000 vertices.

very high. For example, a break through of over than 0.99 reduction in the total communication volume is achieved for application graphs with 3000 vertices, when they are mapped to 16, 32, and 64 processors. Similarly, we have achieved a break through in the reduction of communication volume for application graphs with 6000 vertices, when they are mapped to 16, 32, 64, and 128 processors. In general, a conservative assessment of our VHEM method is that it can produce efficient clustering of application graph models when the ratio of the application graph size to the number of processors is greater than 45 times. Accordingly, it is expected that effective clustering for system models with 256 and 512 processors can be achieved with application graph sizes exceeding 12000 and 24000 vertices, respectively. It is to be noted that the execution cost of the VHEM method maintains the same level of overhead over the HEM method irrespective of the breaking through points.

We have adopted the policy of remapping a border vertex to the best processor on the shortest path to one of its neighboring processors, that would reduce the application's execution time. Figure 2(a) shows the execution time of the refinement phase versus the number of processors. The plots

indicate, implicitly, the effect of the number of tested border vertices for migration on the execution time. The results of the refinement phase on reducing the application time are shown by the plots of the speed-up factor versus the number of processors in Figure 2(b). It is noted that little speed-up can be achieved when the computation weights of partitions are very heavy. This situation can mainly occur when the refinement phase is applied after using the VHEM method, in which a break through in the reduction of the communication volume of the coarsened graph has been achieved. The same effect of the VHEM clustering technique on the refinement phase can be seen from the low execution time at the number of processor, which caused the break through in the reduction of the communication volume.

5. Conclusion

This work has introduced a multilevel graph partitioning paradigm for mapping parallel application tasks to heterogeneous computing resources on the Grid. The contribution of the work is focused on three aspects. In the first, the paradigm considers the dynamic relationship between the application clients and the system's providers on the Grid environment, through the requirements provided by each side. This consideration is reflected in utilizing the client and the system specification to determine the objective function and the constraints of the mapping problem. In the second, the paradigm introduces an efficient heuristic for the coarsening step, called the VHEM method. The simulation results show that the heuristic can achieve very high reduction factor in the communication volume, when the ratio of the number of tasks to the number of processors exceeds a threshold value, without extra overhead in the execution time. In the third, the paradigm introduces an efficient heuristic for the refinement phase for remapping border vertices to other processors. The heuristic uses a space of processor preference for migration that includes the subset of processors on the shortest paths from the currently allocated processor for a vertex to all other processors to which adjacent vertices are allocated. In general, the future work should concentrate on enhancing the initial mapping phase and the refinement phase, and should develop procedures to reduce the effect of a greedy technique for the coarsening step, like VHEM, on the effectiveness of the refinement process.

References

[1] Berman, F., Wolski, H., Casanova, H., et al., "Adaptive Computing on the Grid Using AppLeS," *IEEE Trans. Parallel and Distributed Systems*, vol. 14, no. 4, April 2003, pp. 369-382.

[2] Bokhari, S. H., "On the Mapping Problem," *IEEE Trans. Computers*, vol. c-30, vo. 3, March 1981.

[3] Coope, K., Dasgupta, A., Kennedy, K., et al. "New Grid Scheduling and Resechduling Methods in the GrADS Project," *Workshop for Next Generation Software*, Santa Fe, New Mexico, April 2004.

[4] Foster, I., Kesselman, C., The Grid: Blueprints for a New Computing Infrastructure, Second Edition, Elsevier Inc., 2004.

[5] Karypis, G., and Kumar, V., "Multilevel k-way Partitioning Scheme for Irregular Graphs," *Journal of Parallel and Distributed Computing*, vol. 48, no. 1, 1998, pp. 96-129.

[6] Kumar, S., Das, S. K., and Biswas, R., "Graph Partitioning for Parallel Applications in Heterogeneous Grid Environments," *Proceedings of the IEEE 16th International Parallel and Distributed Processing Symposium (IPDPS 2002)*, Fort Lauderdale, Florida, April 15-17, 2002.

[7] Walshaw, C., and Cross, M., "Multilevel Mesh Partitioning for Heterogeneous Communication Networks," *Future Generation Computer Systems*, vol. 17, 2001, pp. 601-623.

Science Experimental Grid Laboratory (SEGL) Dynamical Parameter Study in Distributed Systems

Natalia Currle-Linde[a], Uwe Küster[a], Michael Resch[a], Benedetto Risio[b]

[a]High Performance Computing Center Stuttgart (HLRS), University of Stuttgart, Nobelstrasse 19, 70569 Stuttgart, Germany

[b]RECOM Services, Nobelstrasse 15, 70569 Stuttgart, Germany

Currently, numerical simulation using automated parameter studies is already a key tool in discovering functional optima in complex systems such as biochemical drug design and car crash analysis. In the future, such studies of complex systems will be extremely important for the purpose of steering simulations. One such example is the optimum design and steering of high power furnaces of power plants. The performance of today's high performance computers and PC-clusters enables simulation studies with results that are as reliable as those obtained from physical experimentation. Recently, Grid technology has supported this development by providing uniform and secure access to computing resources over wide area networks (WANs), making it possible for industries to investigate large numbers of parameter sets using sophisticated simulations. However, the large scale of such studies requires organized support for the submission, monitoring, and termination of jobs, as well as mechanisms for the collection of results, and the dynamic generation of new parameter sets in order to intelligently approach an optimum. In this paper, we describe a solution to these problems which we call Science Experimental Grid Laboratory (SEGL). The system defines complex workflows which can be executed in the Grid environment, and supports the dynamic generation of parameter sets. It also allows the execution of sets of independent tasks of interdependent jobs which can run either synchronously or asynchronously on heterogeneous systems. The automatic collection of results is based on an object-oriented database design.

1. Introduction

During the last 20 years the numerical simulation of engineering problems has become a fundamental tool for research and development. In the past, numerical simulations were limited to a few specified parameter settings. Expensive computing time did not allow for more. More recently, computer clusters with hundreds of processors enable the simulation of complete ranges of multi-dimensional parameter spaces in order to predict an operational optimum for a given system. Testing the same program in hundreds of individual cases may appear to be a straightforward task. However, the administration of a large number of jobs, parameters and results poses a significant problem. An effective mechanism for the solution of such parameter problems can be created using the resources of a Grid environment. This paper, furthermore proposes the coupling of these Grid resources to a tool which can carry out the following: generate parameter sets, issue jobs in the Grid environment, control the successful operation and termination of these jobs, collect results, and generate new parameter sets based on previous results in order to approach a functional optimum, after which the mechanism should gracefully terminate. We expect to see the use of parameterized simulations in many disciplines. Examples are drug design, statistical crash simulation of cars, airfoil design, power plant simulation by varying burners and fuel quality. The mechanism proposed here offers a unified framework for such large-scale optimization problems in design and engineering.

1.1. Existing tools for parameter investigation studies

Tools like Nimrod [1] and Ilab [1] enable parameter sweeps and jobs, running them in a distributed computer environment (Grid) and collecting the data. ILab also allows the calculation of multi-parametric models in independent separate tasks in a complicated workflow for multiple stages. However, none of these tools is able to dynamically generate new parameter sets by an automated optimization strategy. In addition to the above mentioned environments, tools like Condor [1], UNICORE [3] or AppLeS [1] can be used to launch pre-existing parameter studies using distributed resources. These, however, give no special support for dynamic parameter studies.

1.2. Workflow

Realistic application scenarios become increasingly complex due to the necessary support for multiphysics applications, preprocessing steps, postprocessing filters, visualization, and the iterative search in the parameter space for optimum solutions. These scenarios require the use of various computer systems in the Grid, resulting in complex procedures best described by a workflow specification. The definition and execution of these procedures requires user-friendly workflow description tools with graphical interfaces, which support the specification of loops, test and decision criteria, synchronization points and communication via messages. Several Grid workflow systems exist. Systems such as Triana [4] and UNICORE, which are based on directed acyclic graphs (DAG), are limited with respect to the power of the model; it is difficult to express loop patterns, and the expression of process state information is not supported. On the other hand, workflow-based systems such as GSFL [6], and BPEL4WS [6] have solved these problems but are too complicated to be mastered by the average user. With these tools, even for experienced users, it is difficult to describe non-trivial workflow processes involving data and computing resources. The SEGL system described here aims to overcome these deficiencies and to combine the strengths of Grid environments with those of workflow oriented tools. It thus provides a visual editor and a runtime workflow engine for dynamic parameter studies.

1.3. Dynamic parameterization

Complex parameter studies can be facilitated by allowing the system to dynamically select parameter sets on the basis of previous intermediate results. This dynamic parameterization capability requires an iterative, self-steering approach. Possible strategies for the dynamic selection of parameter sets include genetic algorithms, gradient-based searches in the parameter space, and linear and nonlinear optimization techniques. An effective tool requires support of the creation of applications of any degree of complexity, including unlimited levels of parameterization, iterative processing, data archiving, logical branching, and the synchronization of parallel branches and processes. The parameterization of data is an extremely difficult and time-consuming process. Moreover, users are very sensitive to the level of automation during application preparation. They must be able to define a fine-grained logical execution process, to identify the position in the input data of parameters to be changed during the course of the experiment, as well as to formulate parameterization rules. Other details of the parameter study generation are best hidden from the user.

1.4. Databases

The storage and administration of parameter sets and data for an extensive parameter study is a challenging problem, best handled using a flexible database. An adequate database capability must support the *a posteriori* search for specific behavior not anticipated in the project. In SEGL the automatic creation of the project and the administration of data are based on an object-oriented database (OODB) controlled by the user. The database collects all relevant information for the

realization of the experiment, such as input data for the parameter study, parameterization rules and intermediate results. In this paper we present a concept for the design and implementation of SEGL, an automated parametric modeling system for producing complex dynamically-controlled parameter studies.

2. System Architecture and Implementation

Figure 1 shows the system architecture of SEGL. It consists of three main components: the User Workstation (Client), the ExpApplicationServer (Server) and the ExpDBServer (OODB). The system operates according to a Client-Server-Model in which the ExpApplication Server interacts with remote target computers using a Grid Middleware Service. The implementation is based on the Java 2 Platform Enterprise Edition (J2EE) specification and JBOSS Application Server. The System runs on Windows as well as on UNIX platforms. The OODB is realized using the Java Data Objects (JDO) implementation of FastObjects [5].

The client on the user's workstation is composed of the ExpDesigner and the ExpMonitorVIS. The ExpDesigner is used to design, verify and generate the experiment's program, organize the data repository and prepare the initial data. The ExpMonitorVIS is generated for visualization and for the actual control of the complete process. The ExpDesigner allows to describe complex experiments using a simple graphical language. Each experiment is described at three levels: control flow, data flow and data repository. The control flow level is used for the description of the logical schema of the experiment. On this level the user makes a logical connection between blocks: direction, condition and sequence of the execution of blocks. Each block can be represented as a simple parameter study.

The data flow level is used for the local description of interblock computation processes. The description of processes for each block is displayed in a new window. The user is able to describe:

(a) Both a standard computation module and a user-specific computation module. The user-specific module can be added to suit the application domain.

(b) The direction of input and output data between the metadata repository and the computation module.

(c) The parameterization rules for the input set of the data.

(d) Synchronization of interblock processes.

On the data repository level, a common description of the metadata repository is created. The repository is an aggregation of data from the blocks at the data flow level. Each block contains one or more windows representing part of the data flow. Also described at the data repository level are the key and service fields (objects) of the data base.

After completion of the design of the program at the graphical icon-level, it is "compiled". During the "compilation" the following is created:

(a) a table of the connections between program objects on the data flow level for each block (manipulation of data) and

(b) a table of the connections between program blocks on the control flow level for the experiment.

Parallel to this, the experiment's database aggregates the data base icon objects from all blocks / windows at the data flow level and generates query-language (QL) descriptions of the experiment's database. The container application of the experiment is transferred to the ExpApplicationServer and the QL descriptions are transferred to the server data base. Here, the metadata repository is created. The ExpApplicationServer consists of the ExpEngine, the Task, the ExpMonitorSupervisor and the ResourceMonitor.

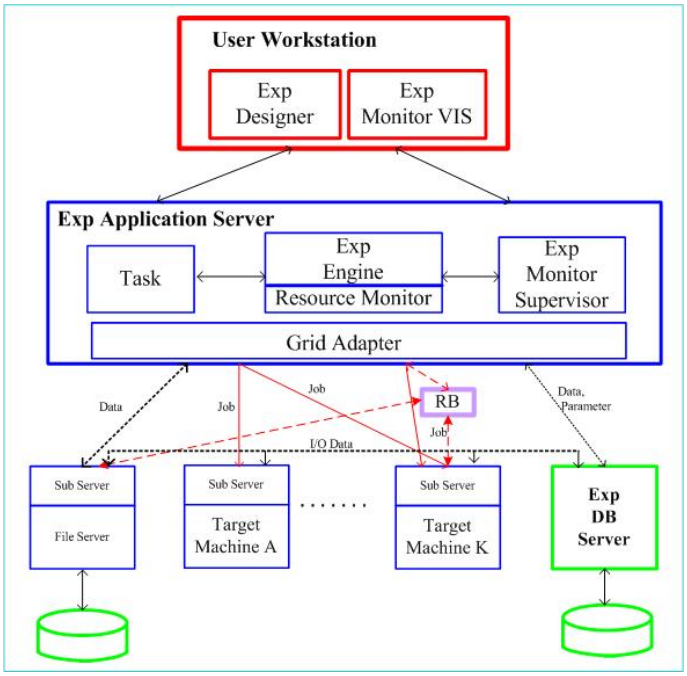


Figure 1: System Architecture

The Task is the container application. The ResourceMonitor holds information about the available resources in the Grid environment. The MonitorSupervisor controls the work of the runtime system and informs the Client about the current status of the jobs and the individual processes. The ExpEngine is the controlling subsystem of SEGL (Runtime subsystem). It consists of three subsystems: the TaskManager, the JobManager and the DataManager. The TaskManager is the central dispatcher of the ExpEngine coordinating the work of the DataManager and the JobManager:

(1) It organizes and controls the sequence of execution of the program blocks. It starts the execution of the program blocks according to the task flow and the condition of the experiment program.

(2) It activates a particular block according to the task flow, chooses the necessary computer resources for the execution of the program and deactivates the block when this section of the program has been executed.

(3) It informs the MonitorSupervisor about the current status of the program.

The DataManager organizes data exchange between the ExpApplicationServer and the FileServer and between the FileServer and the ExpDBServer. Furthermore, it controls all parameterization processes of input data. The JobManager generates jobs and places them in the corresponding SubServer of the target machines. It controls the placing of jobs in the queue and observes their execution.

The final component of SEGL is the data base server (ExpDBServer). All data which occurred during the experiment, initial and generated, are kept in the ExpDBServer. The ExpDBServer also hosts a library tailored to the application domain of the experiment. For the realization of the data base we choose an object-oriented database because its functional capabilities meet the requirements of an information repository for scientific experiments. The interaction between ExpApplicationServer and the Grid resources is done through a Grid Adaptor. Currently, e.g. Globus[2] and UNICORE offer these services.

3. Parameter Modeling from the user's view

Figure 2 shows an example of a task flow for an experiment as it appears in the ExpDesigner. The graphical description of the application flow has two purposes: first, it is used to collect all information for the creation of the experiment and, second, it is used for the visualization of the current experiment in the ExpMonitorVIS. For instance, the current point of execution of a computer process is highlighted in a specific color within a running experiment.

3.1. Control Flow level

Within the control flow (see Figure 2) the user defines the sequence of the execution of the experimentâs blocks. There are two types of operation block: control block and solver block. The solver block is the program object which performs some complete operation. The standard exam-

ple of the solver block can be a simple parameter sweep. The control block is the program object which allows the changing of the sequence of execution operation according to a specified criterion.

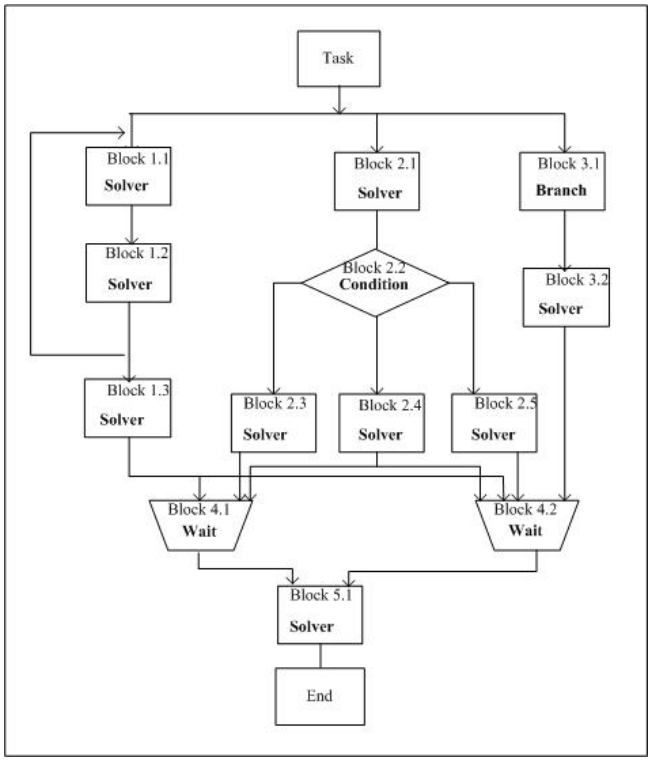


Figure 2: Sample Task Flow (control flow)

Figure 2 shows an example of task flow. After execution of "Task" block 1.1, block 2.1 and block 3.1 are activated simultaneously. In each of these blocks a process is executed. After having worked with the first set of data in block 1.1, the first process in block 1.2 is activated. After execution of the first process in block 1.2, the first process in block 1.3 and the second process in block 1.1 are started according to the logic of the experiment. The input data for the second and the following processes in block 1.1 are prepared in block 1.2 and so on.

3.2. Data Flow level

Figure 3 presents an example of a solver block (Block 1.1). At this level, the user can describe the manipulation of data in a very fine grained way. The solver block consists of computation (C) , replacement (R), parameterization (P) modules and a data base. These are connected to each other with arrowed lines showing the direction of data transfer between modules and the sequence of execution during the computation process.

Each module is a Java object, which has a standard structure and consists of several sections. For example: each computation module (C) consists of four sections. The first section organizes the preparation of input data. The second generates the job and controls its execution. The third initializes and controls the record of the result in the experiment data base. The fourth section controls the execution of module operation. It also informs the main program of the block about the manipulation of certain sets of data and when execution within a block is complete.

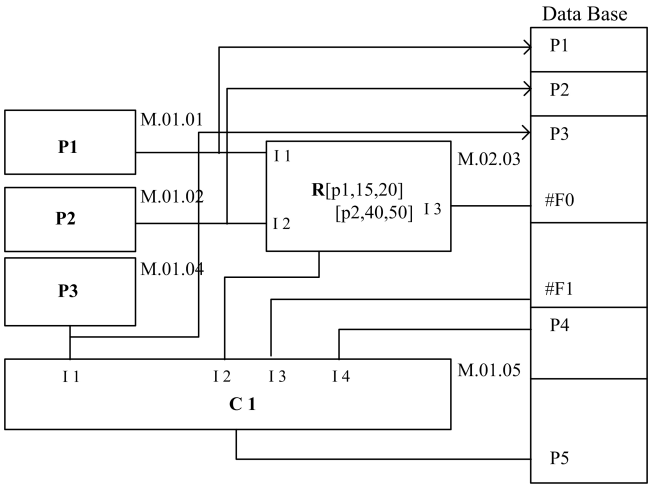


Figure 3: Solver Block 1.1 (data flow)

After a block is started, the parameterization module (P) and replacement module (R) wait for the request from the corresponding inputs of the computation module (C). After that, they generate a set of input data according to rules specified by the user, either as mathematical formulae or a list of parameter values. In this example three variants of parameterization are represented:

(a) Direct transmission of the parameter values with the job. In this case, parameterization module (P3) transfers the generated parameter value to the computation module (C1) upon its request. The computation module generates the job, including converting parameter values into corresponding job parameters. This

method can be used if the parameterized value is a number, symbol or combination of both.

(b) Parameterized objects are large arrays of information (DB-P4 in Figure 3) which are kept in the experiment data base. These parameters are copied directly from the experiment data base to the corresponding file server and then written with the same array name with the index of the number of the stage. In this case, attributes of the job are sent to the file server as references (an array of data).

(c) If it is important, then the preparation of the data is moved outside of the main program. This allows the creation of a more universal computation module. Furthermore, it allows scaling, i.e. avoiding limitations in the size, position, type and number of the parameterized objects used in a module.

In these cases the replacement module is used. During the preparation of the next set of input data, new parameter values P1 und P2 are generated. The generated parameter set is linked with replacement processes and then delivered to the corresponding FileServer, where the replacement process is executed.

After the replacement of the specified parameters, the input data is ready for the first stage of computation. Computation module C1 sends a message to the JobManager to prepare the job for the first stage. The JobManager chooses the computer resources currently available in the network and starts the job. After confirmation from the corresponding SubServer of the Target Machine that the job is in a queue, the preparation of the next set of data for the next computation stage begins. Each new stage carries out the same processes as the previous stage. At all stages, the output file is archived immediately after being received by the experiment's database. The control of all processes takes place according to the pattern described above. After starting the ExpMonitorVIS on their workstation, the user receives continuously updated status information regarding the experiment's progress.

4. Use case: Power plant simulation by varying burners and fuel quality

The liberalization of the energy markets puts more and more pressure on the competitiveness of power companies throughout the world. In order to maintain their competitive edge, it is necessary to optimize the operation of existing power plants towards minimum operational costs. Potential optimization targets can be minimization of excess air (increasing efficiency) or NOx-emission (reducing DeNOx operation costs). Pure experimental optimizations without computer-aided techniques are time-consuming and require a significantly higher manpower effort. Furthermore, in the case of necessary design changes the technical risks involved in the investment decision can only be assessed with computer-aided techniques. Computer-aided methods are well accepted in the power industry. The optimization procedure applied by SEGL for the present problem is based on a genetic algorithm (GA).

In order to work on boiler optimization problems with SEGL, the parameters that have to be optimized are coded in binary form and assembled to a so-called "chromosome". The chromosome carries all the important properties to be changed of the so-called "individuals". A certain number of these artificial individuals are generated initially, the so-called "population", and the GA of SEGL imitates the natural evolution process. The imitation is done by applying the genetic mechanisms Selection, Recombination and Mutation. The basic workflow can be described as follows:

1. Binary coding of optimization parameters and chromosome assembly.
2. Generation of an initial population.
3. Decoding of the chromosome information for each individual.
4. Simulation of the decoded set of optimization parameters with the 3D-furnace simulation code RECOM-AIOLOS for each individual. This is the time consuming step.

5. Filtering the 3-D results of the furnace simulation to derive the target values for each individual.

6. Evaluation of the performance level for each individual (terminate the optimization process if desired optimization level is reached).

7. Selection of suitable individuals for reproduction and Recombination/Mutation of the chromosome information for the selected individuals to generate new individuals.

8. Return to Step 3 for new individuals.

4.1. Industrial Applicability

An experimental operation optimization exercise performed in 1991 at a power station in Italy (ENEL's coal-fired Fusina) is used to demonstrate the capabilities of SEGL. In a windbox, the amount of air flowing through a nozzle is controlled by the damper setting of the nozzle. A damper setting of 100% means that the flow passage of the nozzle is fully open. Reducing the damper setting of a single nozzle allows the reduction of the air mass flow through the nozzle, but at the same time the air mass flows for all other nozzles in the windbox are increased. In 1991 separate overfire air nozzles (separate OFA) were installed above the main combustion zone to minimize NOx-emissions. A new operation mode was required after the successful installation of the separate overfire air to maintain the lowest possible NOx-emission together with a minimum unburned carbon loss. In 1991 this optimization exercise was solved experimentally. In a series of 15 tests over a duration of approximately 10 days, 15 operation modes were tested with varying amounts of close coupled overfire air (CCOFA), separate OFA, and tilting angle of the separate OFA ($\pm 30^o$).

The following operation experience was recorded to identify an optimized operation:

a) For a horizontal orientation of the separate OFA the maximum NOx-reduction is reached with dampers 100% open.

b) A tilting of the separate OFA to -30^o has a minor effect on the NOx-emission but improves the burnout (reduced unburned carbon loss).

c) A tilting of the separate OFA to $+30^o$ leads to an NOx-reduction but increases the unburned carbon loss significantly.

d) Closing the CCOFA completely at 100% open separate OFA has only a minor effect on the NOx-emission.

In order to work on this combustion optimization problem in virtual reality, a high-resolution boiler model with 1 Mio. grid points and a reduced boiler model with only 200,000 grid points was generated. In order to reduce the computational effort, the optimization environment works only with the reduced boiler model. The spatial resolution of the reduced boiler model was reduced to a degree that still allows qualitatively accurate predictions. For the optimized settings that are identified during the automatic optimization, a simulation run on the high-resolution model is required to generate quantitatively reliable answers (see Table 1). As shown in Table 1, an accuracy of approximately $\pm 10\%$ between simulation and reality can be reached on the high-resolution boiler model. The optimization parameters "OFA damper setting", "CCOFA damper setting", and "Tilting Angle" were coded with 4 bit on the chromosomes. NOx-emission and C in Ash values achieved in the model were combined to a target function for the evaluation of the individuals. The underlying combined evaluation target function is

Target Function = Evaluation [NOx] + Evaluation [C in Ash].

The GA required approximately 11 generations with 10 individuals per population to identify an optimized parameter set. During the course of the automatic optimization, approximately 51 of the entire 4096 ($2^4 \cdot 2^4 \cdot 2^4$) coded combinations of parameter settings were evaluated with respect to the target functions. Table 2 shows the development of the best individuals in each generation in the course of the automatic optimization. The results demonstrate that SEGL is able to identify

Table 1
Measured and calculated (high-resolution) NOx-emission and C in Ash

	NOx-emission $[mg/m_n^3, 6\%O2]$		C in Ash [%]	
Setting	measured	calculated	measured	calculated
No OFA No CCOFA	950 - 966	954	6.41 - 7.50	5.66
No OFA CCOFA: 100%	847 - 858	794	7.47 - 7.61	6.58
OFA:100% CCOFA: 100%	410 - 413	457	10.43 - 11.48	10.28

the same positive measures that were found in the experimental optimization. The final run on the high-resolution boiler model led to an NOx-emission of $476mg/m_n^3$ at $6\%O_2$ and a C in Ash value of 8.42 %. Both values are in the range of the emission and C in Ash values that were observed in the field after the optimization exercise. The total duration of the automatic optimization was only 3.5 days on a high performance vector-computer.

Table 2
Development of best individuals in each generation during automatic optimization

Generation	Target-Value	OFA [%]	CCOFA [%]	Tilting Angle $[^o]$	NOx mg/m_n^3	C in Ash[%]
Basis	12.070	0	0	0	805	3.39
1	10.061	100	100	-30	479	10.84
5	9.600	93	93	-30	473	10.42
10	9.177	93	20	-30	458	10.26

5. Conclusion

This paper presented the concept and description of the implementation of SEGL for the design of complex and hierarchical parameter studies which offers an efficient way to execute scientific experiments. We can show that SEGL allows to substantially reduce optimization costs for parameter studies.

References

[1] de Vivo, A., Yarrow, M. McCann, K.: A comparison of parameter study creation and job submission tools; Technical report NAS-01002, NASA Ames Research Center, Moffet Filed, CA, 2000

[2] Foster, I., Kesselman C.: The Globus Project: A Status Report; In: Proc. IPPS/SPDP'98 Heterogeneous Computing Workshop, pp4-18, 1998.

[3] Erwin, D. (Ed.): Joint Project Report for the BMBF Project UNICORE Plus, Grant Number: 01 IR 001 A-D, Duration: January 2000 - December 2002;

[4] Taylor, I., Shields, M., Wang, I., and Philp, R.: Distributed P2P Computing within Triana: A Galaxy Visualization Test Case; IPDPS 2003 Conference, Nice / France, 2003.

[5] FastObject webpage: http://www.fastobjects.com

[6] Tony, A., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., Weerawarana, S.: Specification: Business Process Execution Language for Web Services Version 1.1, May 05, 2003: http://www.106.ibm.com/developerworks/library/ws-bpel

On Scheduling in UNICORE – Extending the Web Services Agreement based Resource Management Framework*

A. Streit[a], Oliver Wäldrich[b], Ph. Wieder[a], W. Ziegler[b]

[a]Central Institute for Applied Mathematics, Research Centre Jülich, 52425 Jülich, Germany

[b]Fraunhofer Institute SCAI, Department of Bioinformatics, 53754 Sankt Augustin, Germany

1. Introduction

Designing, building and using Grids generally implies that the resources involved are highly distributed, heterogeneous and managed by different organisations. These characteristics hamper the coordinated usage and management of resources within a Grid, which in turn motivates the development of Grid-specific resource management and scheduling solutions. Please refer for instance to [8] which collects a large variety of contributions to research and development for this specific topic.

The use case of the work we present here requires a Grid that has exactly the properties listed above: The VIOLA Grid [15] comprises distributed resources of different types which are owned by different organisations and have to be managed in a coordinated fashion. Although the resources have different owners the aspects of authentication and authorisation are of secondary importance to this work since the underlying middleware, UNICORE, provides all necessary security means required to realise a Virtual Organisation [3].

The focal point of this paper is the orchestration of resources. In our case various compute and network resources have to be co-allocated to form a virtual machine that enables the execution of distributed parallel applications. To achieve this goal, a meta-scheduler is needed which generates a schedule based on user requirements and guarantees that the requested resources are reserved in advance. Therefore we developed the VIOLA MetaScheduling Service, which is founded on the experience with the UNICORE WS-Agreement prototype [11], the MeSch meta-scheduler [10] and the UNICORE timer process [2, chapter 3.7.2 “Distributed Applications”]. The integration of UNICORE and the MetaScheduling Service provides a framework which fulfils the use case’s requirements and lays the foundation for co-allocation of arbitrary resources.

Following the introduction (this section) we describe in Section 2 the resource management mechanisms of the UNICORE software and the resulting challenges with respect to the VIOLA use case. Based on this description we introduce in Section 3 the VIOLA MetaScheduling Service including its architecture and the internal processes. We conclude the paper with an outline of future work related to scheduling in UNICORE (see Section 4) and a short summary (see Section 5).

2. UNICORE Resource Management

The resource management in UNICORE is centred around two concepts: resource virtualisation and job abstraction [2]. These concepts are connected through the *UNICORE Resource Model*, which provides a unified mechanism to describe both resource requests and resource properties.

The virtualisation of resources manifests itself in UNICORE’s *Virtual Site* (Vsite) that comprises

*This work is partially funded by the German ministry for eduction and research (BMBF) within the VIOLA project under grant 01AK605L. This research work is also partially carried out under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

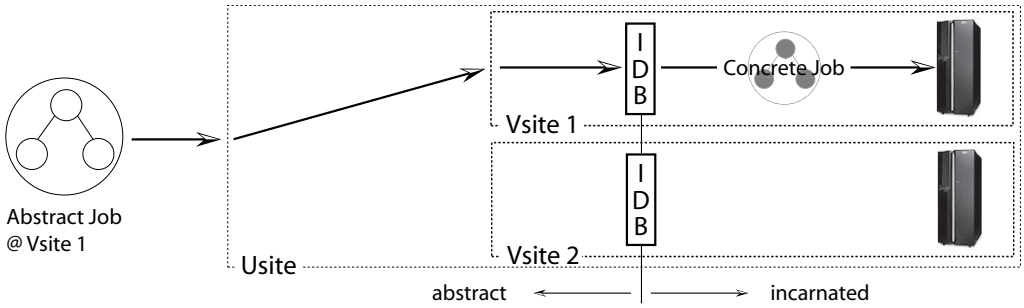


Figure 1. Two stages of a UNICORE job

a set of resources. These resources must have direct access to each other, a uniform user mapping, and they are generally under the same administrative control. A set of Vsites is represented by a *UNICORE Site* (Usite) that offers a single access point (a unique address and port) to the resources of usually one institution.

A UNICORE job passes two different stages during its lifetime (see Fig. 1), an abstract and an incarnated one. During the former the job description contains no site-specific information such as paths to executables or paths to data, and it is processable by every Usite, whereas during the latter the job representation includes all information necessary to actually execute the job on resource level. Between these two stages the abstract job representation is "translated"; a process that is called *incarnation* and is carried out at Vsite level. For incarnation as well as for the composition of jobs resource-specific information is needed. The information service that keeps and propagates this information is the *Incarnation Database* (IDB), a service deployed per Vsite. It maintains resource information according to the UNICORE Resource Model and provides inter alia information about *capability resources*, such as software distributions, and about *capacity resources*, such as processors or memory.

Most Grid scheduling actions within a UNICORE environment are currently carried out manually. Based on the classification in [12], ten actions in three phases are distinguished during the process of scheduling. Resource discovery and system selection are performed by the user. Entering the third phase called job execution (on Usite level) the abstract job representation has then its target Vsite assigned, as shown in Fig. 1. This implies that candidate resources have been pre-selected and mapped to the user's or application's resource requirements. In the current implementation these tasks are executed by the user via the UNICORE graphical client although the UNICORE concepts and models do not impose this modus operandi.

To satisfy the VIOLA meta-scheduling use case UNICORE needs to be at least extended to provide advance reservation capabilities. This can be realised by integrating WS-Agreement providers and consumers into UNICORE, as reported in [11], and has been implemented in the first phase of the MetaScheduling Service development as outlined in the following section.

3. Scheduling Arbitrary Resources

Scheduling resources with the objective of meeting a user's demand for a distinct Quality of Service (QoS), e.g. precisely timed availability of multiple resources, requires local scheduling systems with two indispensable capabilities in order to allocate resources efficiently:

- To publish the time-slots at which the resource is available.
- To make a reservation for a given start time and stop time.

At this stage there are just a few local scheduling systems with these capabilities, such as PBS Professional [9], CCS [7], EASY [13], or LSF. But we expect new systems appear (like for example the next version of LoadLeveler) as these capabilities are crucial for resource owners who intend to advertise their resources with guarantees for QoS to the Grid. Other approaches have been proposed to support co-allocation of resources based on simple queueing by the local scheduling systems. Following either a trial and error approach or extending the reservation times hoping to create a matching time-window for the job, these approaches deliver only best-effort results. The first approach keeps the local scheduling busy with requests and cancellation of requests, while the second one often results in wasting resources that are waiting for other resources to become available if the local queues are populated.

3.1. Interaction with Local Scheduling Systems

Local scheduling systems typically have different APIs with different behaviour. The handling of these differences is not part of the MetaScheduling Service, instead an Adapter Pattern [4] approach is used. This results in a specific light-weight adapter for each local scheduling system which is designed to facilitate the integration of new local scheduling systems. The adapters implement the (command line) interface to the local scheduling system on one end and the SOAP [14] interface to the MetaScheduling Service on the other end. The MetaScheduling Service itself is agnostic to the details of local scheduling systems and implements the SOAP interface which includes all functions necessary to carry out meta-scheduling:

- `ResourceAvailableAt` - returns the next possible start time for a job on a resource that matches the properties of a user's request.
- `Submit` - submits a job to a free slot identified before.
- `Cancel` - removes a component of a meta-job from a local scheduler.
- `State` - returns the state of a local reservation.
- `Bind` - submits IP addresses of allocated resources (at runtime).
- `Publish` - submits runtime information to the local system.

Only the first four functions are needed to generate a meta-schedule, the remaining two provide additional management functions. In case of network resources the `Bind` call allows to communicate the IP addresses of all the compute nodes which are allocated for a job once the job has been started. Thus the resource management system for the network resources is enabled to set up the necessary end-to-end network configuration with the required QoS. The `Publish` call finally is used to submit runtime information to the local systems, like information necessary to configure the MPI environment or wrappers for batch scripts to start-up a job.

The pseudo code in Listing 1 describes the algorithm to identify common free slots using the `ResourceAvailableAt` call. To speed up the process the `ResourceAvailableAt` calls are executed in parallel, hence the request for the next possible start time is performed in parallel for all adapters.

Listing 1: Pseudo code of the time-slot algorithm

```
set n = number of requested resources
set resources[1..n] = requested resources
set properties[1..n] = requested property per resource # number of nodes, bandwidth, time,...
set freeSlots[1..n] = null                             # start time of free slots
set endOfPreviewWindow = false
set nextStartupTime = currentTime+someMinutes          # the starting point when looking for free slots

while (endOfPreviewWindow = false) do {

  for 1..n do in parallel {
    freeSlots[i] =  ResourceAvailableAt(resources[i], properties[i], nextStartupTime)
  }

  for 1..n do {
    set needNext = false
    if (nextStartupTime != freeSlots[i]) then {
      if (freeSlots[i] != null) then {
        if (nextStartupTime < freeSlot[i]) then {
          set nextStartupTime = freeSlots[i]
          set needNext = true
        }
      } else {
        set endOfPreviewWindow = true
      }
    }
  }
}

if ((needNext = false) & (endOfPreviewWindow = false)) then return
  freeSlots[1] else return "no common slot found"
```

3.2. Capabilities of a MetaScheduling Service

There are several tasks a meta-scheduler should be able to perform in a resource management framework. It should for example be able to:

1. Allocate a single resource for a single application for a fixed period of time.
2. Co-allocate several resources for the same fixed period of time for single or multiple applications.
3. Allocate multiple resources for multiple applications for different fixed periods of time.
4. Allocate dedicated resources for either of the cases above.

The current version of the MetaScheduling Service is able to perform task 1. and 2. which allow to schedule a simple job to one resource in the Grid or to schedule a job composed of several components that need to be executed at the same time to multiple resources in the Grid. The third task, being essential to schedule workflows in the Grid without wasting resources, will be implemented in the next version of the service, just as the fourth task which is required to allocate resources with special capabilities, i.e. I/O nodes for parallel I/O.

3.3. Negotiation Process

In this section we describe the protocol the MetaScheduling Service uses to negotiate the allocation of resources with the local scheduling systems. The protocol is illustrated in Fig. 2. As Section 3.1 describes, the resource negotiation starts with the MetaScheduling Service acquiring a preview of the resources available in the near future from the individual scheduling subsystems. The consolidation of the previews from all systems results in a list of resources and the next possible start times for the job scheduled to these resources.

The algorithm terminates either if at least one resource no longer advertises a next possible start time within the look-ahead time-frame of the local scheduling system, or when a common time-slot is identified. As long as all resources return the next possible start times, the MetaScheduling Service calculates whether there is sufficient overlap between the returned slots to allocate all requested

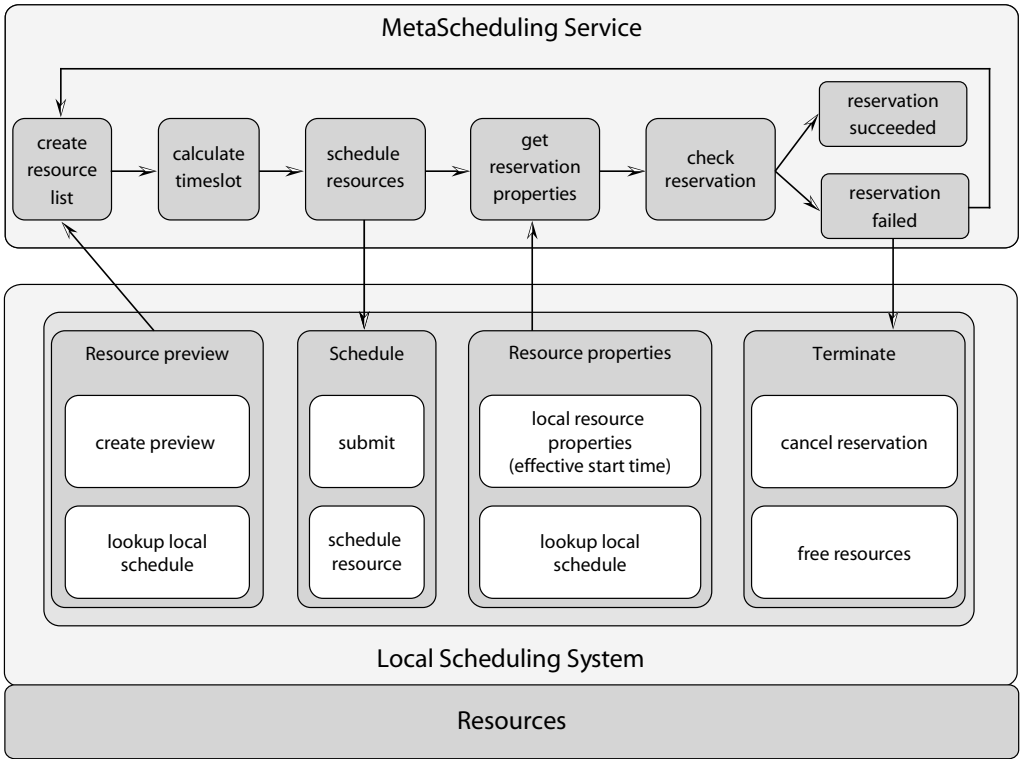


Figure 2. The negotiation process

resources. If not, the slot with the latest possible starting time is selected as starting point of the next preview query. If there is sufficient overlap, the next steps of the protocol are performed:

- The MetaScheduling Service submits the specific reservation requests with the calculated start time to the local scheduling systems.
- The adapters return a confirmation from the local scheduling systems, indicating the start time of the reservation and the properties (number of nodes, bandwidth, etc.).
- The MetaScheduling Service then checks whether the confirmations received match the requests.

Verifying the schedules is necessary because in the meantime new reservations may have been submitted to the local schedulers by other (local) users or processes, which may prevent the scheduling of the reservation at the requested time. If the MetaScheduling Service detects one or more reservations that are not scheduled at the requested time, all reservations will be cancelled. The latest effective start time of all reservations in the current scheduling cycle will be used as the earliest starting time in the next negotiation cycle. If this finishes successfully and all reservations are scheduled at the appropriate time on the local systems the co-allocation of the resources will be completed.

3.4. Integration into the UNICORE Environment

As shown in Fig. 3 the UNICORE system and the MetaScheduling Service are currently loosely coupled through the UNICORE client. A special plug-in for the UNICORE client allows the user

to specify his request for co-allocated resources to run a distributed job in the VIOLA testbed. The same plug-in also manages communication with the MetaScheduling Service. Once Unicore/GS, the service-oriented version of UNICORE, will be available, the integration will be rather at the level of the UNICORE Server. But even with the current level of integration it is possible to use UNICORE mechanisms to map the certificate a user presents to the client for authentication to the (probably) different uids of this user at the different sites of the testbed. This information is passed to the MetaScheduling Service along with a list of the requested resources and enables the service to do the reservations at the different sites with the correct local uid. The SOAP interface for communication between UNICORE client and MetaScheduling Service is using WS-Agreement [1] to exchange the details and conditions of the reservation:

- The client requests an agreement template from the MetaScheduling Service.
- The MetaScheduling Service replies with the template containing information about the available resources (UNICORE Vsites).
- The service sends an agreement proposal containing the resource request information.
- If the (co-)allocation is successful, the MetaScheduling Service returns the final agreement.

Once the (co-)allocation has been completed with the reception of the final agreement the UNICORE client creates the abstract UNICORE job that contains a request specified by the user, submits the job to the UNICORE gateway and invokes the normal UNICORE processing of a job as described in Section 2.

In the current framework there is neither a resource detection service nor a resource selection service, thus the user has to explicitly specify the resources he wants to use for a job, i.e. sites and corresponding properties of the reservations. In a later version a broker function will be integrated, allowing a user to describe his request by specifying resource properties and leaving the selection of appropriate resources to the broker. This also relieves the user from resource discovery and system selection if he does not want to control these tasks. The MetaScheduling service then may use the selection suggested by the broker to negotiate the co-allocation or other QoS requirements specified by the user.

4. Future Work

The performance of the the MetaScheduling Service – integrated into UNICORE middleware – will be explored in a performance evaluation. For this purpose the distributed test environment within the VIOLA project is used as a basis. Within this test environment several compute resources are integrated, each one equipped with a different resource management system capable of advance reservation. We will use EASY, PBS Professional, and CCS, while the latest version of LoadLeveler Multi-Cluster (LL-MC), which also supports advance reservations, will be integrated as soon as the production version becomes available.

This test environment will be used to evaluate the performance of the MetaScheduling Service architecture with different baseline scenarios, applied workloads, and meta-job requests. Besides measuring the costs of scheduling for placing meta-jobs with advance reservations, additional criteria will be evaluated. For example, the success of the MetaScheduling Service in placing meta-jobs can be measured by the rejection rate for such jobs with given start times, and by the average waiting time for meta-jobs, which should be placed by the MetaScheduling Service as soon as possible after

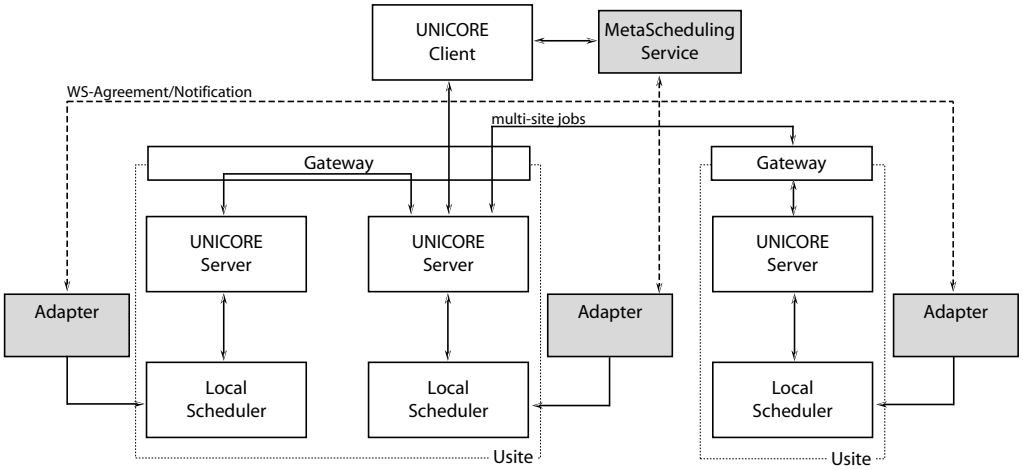


Figure 3. The meta-scheduler architecture

their submission. Additional metrics will be used if meta-jobs are described only by their total size of requested processors and not with a by partitioning. The local schedulers' performance is also influenced by the MetaScheduling Service, as the sub-parts of meta-jobs are placed as advance reservations and thereby influence the local scheduler when scheduling local batch jobs. Hence, the slowdown or average response time of local jobs has to be measured, too.

In addition we will enhance the MetaScheduling Service with Quality of Service and enhanced Service Level Agreements (SLA) functions. In particular negotiating and guaranteeing resource usage will be a key element to bridge the gaps of existing Grid systems as e.g. identified in the NGG2 report [6]. One approach to include such functionality is to apply the planning-based scheduling approach as described in [5]. Queueing-based scheduling considers only the present use of resources. In contrast, planning-based scheduling plans the present and future resource usage by assigning proposed start times to all submitted jobs and updating these whenever the schedule changes. This concept makes it easy to support guaranteed resource usage through advance reservation. The planning-based scheduling approach is implemented in the resource management software CCS [7] and has proved its abilities and benefits in recent years. CCS also provided the basis for an initial proof-of-concept implementation of a meta-scheduler within the UNICORE Plus project [2, chapter 3.7.2 "Distributed Applications"].

5. Conclusion

Coordinated and guaranteed resource provision with QoS and SLA support is a key element of modern resource management and scheduling solutions for Grid environments. UNICORE and its WS-Agreement based resource management framework already provide many desired capabilities. Within the VIOLA project a MetaScheduling Service was developed, which uses advance reservations on local systems for placing multi-site jobs onto the Grid.

In this paper we presented the integration of the MetaScheduling Service into the UNICORE Grid system, work that is carried out to provide meta-scheduling functions for the VIOLA testbed. Details of the implementation and the meta-scheduling processes were given. An even tighter integration with UNICORE will become possible once the Unicore/GS software will be available.

References

[1] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, T. Nakata, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu. Web Services Agreement Specification, July 2005. Online: https://forge.gridforum.org/projects/graap-wg/document/WS-AgreementSpecification/en/16.

[2] D. Erwin, editor. *UNICORE Plus Final Report – Uniform Interface to Computing Resources*. UNICORE Forum e.V., 2003. ISBN 3-00-011592-7.

[3] I. Foster, C. Kesselmann, J. M. Nick, and S. Tuecke. The Pysiology of the Grid. In F. Berman, G. C. Fox, and A. J. G. Hey, editors, *Grid Computing – Making the Global Infrastructure a Reality*, pages 217–249. John Wiley & Sons Ltd, 2003.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[5] M. Hovestadt, O. Kao, A. Keller, and A. Streit. Scheduling in HPC Resource Management Systems: Queuing vs. Planning. In D. G. Feitelson and L. Rudolph, editors, *Proc. of the 9th Workshop on Job Scheduling Strategies for Parallel Processing*, volume 2862 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2003.

[6] K. Jeffery et al. Next Generation Grids 2 – Requirements and Options for European Grids Research 2005-2010 and Beyond, 2004. Online: `ftp://ftp.cordis.lu/pub/ist/docs/ngg2_eg_final.pdf`.

[7] A. Keller and A. Reinefeld. Anatomy of a Resource Management System for HPC Clusters. In *Annual Review of Scalable Computing*. Singapore University Press, 2001.

[8] J. Nabrzyski, J. Schopf, and J. Weglarz, editors. *Grid Resource Management – State of the Art and Future Trends*. Kluwer Academic Publishers, 2004.

[9] PBS Professional. Website. Online: `http://www.altair.com/software/pbspro.htm`.

[10] G. Quecke and W. Ziegler. MeSch – An Approach to Resource Management in a Distributed Environment. In *Proc. of 1st IEEE/ACM International Workshop on Grid Computing (Grid 2000)*, volume 1971 of *Lecture Notes in Computer Science*, pages 47–54. Springer, 2000.

[11] M. Riedel, V. Sander, Ph. Wieder, and J. Shan. Web Services Agreement based Resource Negotiation in UNICORE. In *Proc. of the 2005 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'05)*, pages 31–37. CSREA Press, June 2005.

[12] J. Schopf. Ten Actions When Grid Scheduling – The User as a Grid Scheduler. In J. Nabrzyski, J. Schopf, and J. Weglarz, editors, *Grid Resource Management – State of the Art and Future Trends*, pages 15–23. Kluwer Academic Publishers, 2004.

[13] J. Skovira, W. Chan, H. Zhou, and D. Lifka. The EASY — LoadLeveler API Project. In D. G. Feitelson and L. Rudolph, editors, *Proc. of 2nd Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1162 of *Lecture Notes in Computer Science*, pages 41–47. Springer, 1996.

[14] Simple Object Access Protocol Specification, SOAP Specification version 1.2. Website. Online: `http://www.w3.org/TR/soap12`.

[15] VIOLA - Vertically Integrated Optical Testbed for Large Applications in DFN, 2005. Web site. Online: `http://www.viola-testbed.net`.

A framework for dynamic adaptation of parallel components

Jérémy Buisson[a], Françoise André[b], Jean-Louis Pazat[a]

[a]IRISA/INSA de Rennes, Campus universitaire de Beaulieu, avenue du Général Leclerc, 35042 Rennes CEDEX, France

[b]IRISA/Université de Rennes 1, Campus universitaire de Beaulieu, avenue du Général Leclerc, 35042 Rennes CEDEX, France

Abstract – The emergence of dynamic execution environments such as Grids forces applications to take dynamicity into account. Whereas sudden resource disappearance can be handled thanks to fault-tolerance techniques, these approaches are usually not well suited when resource disappearance is announced in advance. However, this case occurs in particular for resource preemption due to resource sharing or maintenance operations. Similarly, fault-tolerance techniques commonly do not take into account resource appearance. On the other side, dynamic adaptation covers techniques for handling changes in the execution environment. This article presents a framework intended to help developers in the task of designing dynamically adaptable (but not fault-tolerant) components. This article puts the emphasis on an experimental evaluation of the cost of using such a framework.

1. Introduction

The increase of resource consumption by applications is a fact. This is what leads to the introduction of notions such as meta- then grid-computing. Those approaches, that can be coarsely seen as resource pooling, permit to increase significantly the number of resources available to applications. However, increasing the number of resources lowers the mean time between failures. In addition, resource pooling requires users to share the resources; and it prevents them from controlling maintenance operations as they can with their own resources. This makes execution environments dynamic. Applications executed on such environments must take into account the dynamicity of the environment. Otherwise, they would not be able to perform well, and may even not be able to complete.

Dynamic adaptatability is one approach that can be used to tackle the problem of the dynamicity of execution environments. It consists in the ability of applications to modify themselves during their execution according to some observations. If the application observes its execution environment, it thus adapts itself according to its environment.

This article presents a framework for easing building dynamically adaptable components. Section 2 provides a description of dynamic adaptation. Section 3 presents the architectural view of an adaptable component. Section 4 focuses on the problem of coordinating the execution of the adaptation with the execution of the component itself. Section 5 describes the experimental results we obtained with our prototype framework. Section 6 compares our proposal to existing related works.

2. Dynamic adaptation

Dynamic adaptation is an event-based approach for dealing with dynamicity during the execution. When the component observes a change that is significant enough, it decides to react to this change. For example, when a component observes that new processors become available, it may increase its parallel degree by spawning new processes. At an abstract level, dynamic adaptation requires that the component is able to make observations, take a decision and execute a reaction previously

defined. What is exactly observed, how decision is made and which actions must be performed to execute the reaction are closely related to the component and to the goal given to dynamic adaptation. In the given example, the component observes processors because it is able to execute a parallel implementation of itself; the component decides to increase its parallel degree as it aims at executing as fast as possible; it does so by spawning new processes because it is its way of executing on a parallel environment. This shows that dynamic adaptation may not be done without the help of developers. Nevertheless, we can exhibit generic mechanisms and provide developers with a framework for both designing and programming dynamically adaptable components.

3. Framework for dynamic adaptation

The model for dynamic adaptability of software components we defined is divided into several functional "boxes" distributed in three levels as shown on figure 1.

At the functional level, the service provides an implementation of what the component is expected to do. If the component was not dynamically adaptable, it would contain only the service.

The component-independent level contains all mechanisms that can be defined independently of the content of the service functional box. The decider box is the start point of any adaptation. It decides whether the component should be adapted or not. To do so, it relies on incoming events and on some external probes. The connection to the external probes is modeled by the two ports exposed by the decider. The actual trigger of the decision-making process may either result from the reception of an event or be spontaneous. Once the decider has decided that the component should be adapted, it transmits a reaction to the planner. The reaction describes the kind of the adaptation that should be performed. Given this reaction, the planner establishes a plan for applying it. This plan is mostly a collection of action invocations connected by some control flow. This plan is given to the executor, which executes the invocations with respect to the provided control flow. To do so, it relies on coordinator which coordinates the invocations with the execution of the service. As section 4 details, several kinds of coordinators may be used.

The component-specific level is a placeholder for the developer to put specializations of the adaptation framework. The policy permits the developer to specialize the decider for the needs of its component. It describes how decisions can be made. The plan templates describe how the planner can build plans depending on the requested reaction and on the current execution environment. The actions are the elementary tasks that can be invoked from the plans. In order to simplify its task, the developer may be provided with a library of predefined actions. Endly, the relation between the service and the coordinator is weaved thanks to aspect-oriented techniques through the "adaptable"

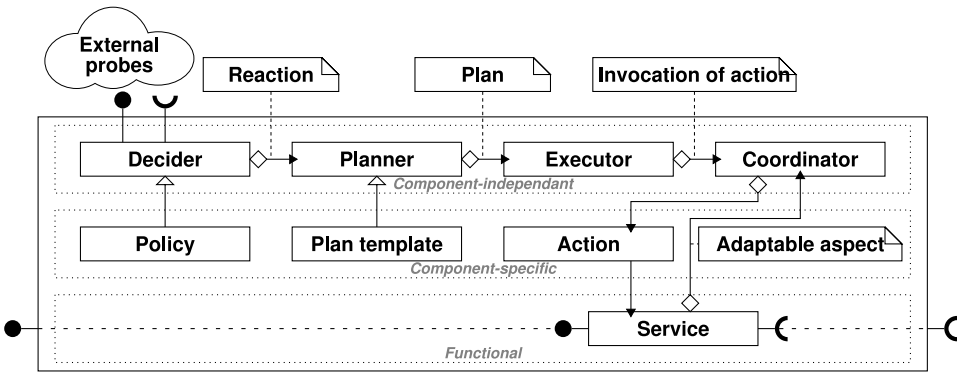


Figure 1. Architecture of an adaptable component

aspect. This aspect is parameterized specifically to the component.

An example of specialization of the framework is given in section 5.1 that describes the demonstrative component used for experiments.

4. Coordination of the invocations of the actions

As it has been previously described in section 3, the purpose of coordinators consists in coordinating the invocations of the actions requested by the plan with the execution of the service. Several coordination policy can be identified and classified according to several concerns:

- Parallel degree. The action may be invoked in a sequential way. When the service contains a parallel code, the action may also be invoked with the same parallel degree as the service functional box.
- Context of execution. The action may be invoked from within the context of the threads of the service. Alternatively, it may be invoked from the context of the processes of the service. It may also be invoked from processes distinct from those of the service, possibly hosted by other machines.
- Synchronization. The action may be invoked asynchronously with regard to the service. It may alternatively require the service to suspend its execution in a special state.

The main purpose of the synchronization concern consists in ensuring that the adaptation does not change (at least semantically) the results produced by the component. Indeed, some actions may not be allowed to be invoked from any state of the service. For example, a matrix redistribution might not be done while the matrix is being modified.

This is why the notion of "point" has been introduced. We call "points" the special states from which actions are allowed to be invoked. Points mostly consists in annotations in the source code of the service functional box; they are instantaneous statements placed by the developer at the locations at which he considers actions can be safely invoked. Given this context, solving the synchronization concern for a coordinator consists in choosing one point at which the action will be invoked.

When the service encapsulates a parallel code, the notion of "point" extends to the global dimension: a "global point" is a collection (one per thread of the service) of points. The choosability of a global point is restricted by a given consistency model. An example consistency model may intuitively assume that the parallel code consists in several parallel steps. Such a model would allow adaptation only between those steps. This model has been described in [2]; a proposal for implementing it has been presented in [3] that restricts the choice to points in the future of the execution path. To do so, it relies on annotations of the code to track the progress of the execution and on a representation of the control flow graph to predict future states. Those annotations are the result of weaving the "adaptable" aspect described in section 3; whereas the points, placed manually by the developer, are the parameters that specialize this aspect for the component.

5. Experiments

The experiments we have made are based upon the NAS Parallel Benchmark [10] 3.1 FFT code for MPI. This code computes the fast fourier transform of a $256 \times 256 \times 128$ matrix from within an iteration. For the purpose of the experiments, it has been slightly modified to use the framework. The modifications are exclusively annotations for indicating adaptation points and for tracking the progress of the execution. Those annotations consist in calls to some functions of the framework.

Experiments have been done using a cluster of dual 2.4 Xeon PC. Each PC hosts at most one process with exactly one thread of the service. For the communications, the service of the component uses LAM-MPI [4]; the framework uses OmniORB as an implementation of CORBA.

5.1. Specialization of the framework for the experiments

For the experiments, the FFT component has been made able to modify its parallel degree depending on the number of machines available in the cluster. The policy is given by figure 2. It states that when new machines become available, the component should spawn new processes; whereas when some machines are announced to disappear, it should stop the corresponding processes.

Algorithm $policy\,()$:
- upon $new\_machines\_appear\,(machines)$:
  $spawn\_process\,(machines)$
- upon $machines\_reclaimed\,(machines)$:
  $terminate\_process\,(machines)$

Figure 2. Policy for adaptable FFT

Figures 3 and 4 show the plan templates for both reactions. The prefix of actions in brackets gives the constraints for each of the coordination concerns listed in section 4.

In order to spawn new processes (reaction $spawn\_process$), the component have to be made available on the corresponding machines (action $deploy\_on$); then the processes must be created (action $spawn\_process\_on$); endly, the matrices have to be redistributed (action $redistribute\_matrices$).

Algorithm $spawn\_process\,(new\_machines)$:
[sequential, in distrinct process, asynchronously] $deploy\_on\,(new\_machines)$
[parallel, in service threads, same point] $spawn\_process\_on\,(new\_machines)$
[parallel, in service threads, same point] $redistribute\_matrices\,()$

Figure 3. Plan template for spawning processes

Similarly, to terminate processes (reaction $terminate\_process$), the component have firstly to redistribute its matrices (action $redistribute\_matrices$); then the processes executed by the reclaimed machines must terminate their execution (action $exit$); endly, everything that was previously installed specifically for those machines has to be cleaned up (action $cleanup$).

Actions are implementations of the invocations requested in plans. Those implementations are dependent on the component and its implementation. For example, the $deploy\_on$ action may transfer files and start required daemons; the $spawn\_process\_on$ action uses of the MPI_Comm_spawn function as the FFT component uses the MPI communication library.

5.2. Timeline of an adaptation

This experiment aims at showing the actions of an adaptation and their timing. This would permit to show how the different phases of dynamic adaptation relates one to each others. This experiment consists in one run of the demonstration component with one adaptation that increases the number of processes from two to four. Figure 5 shows the execution trace near this adaptation.

Algorithm $terminate\_process\,(reclaimed\_machines)$:
  [parallel, in service threads, same point] $redistribute\_matrices\,()$
  if $(local\_machine \in reclaimed\_machines)$ **then**
    [parallel, in service threads, asynchronously] $exit\,()$
    [parallel, in distrinct process, asynchronously] $cleanup\,()$
  end if

Figure 4. Plan template for terminating processes

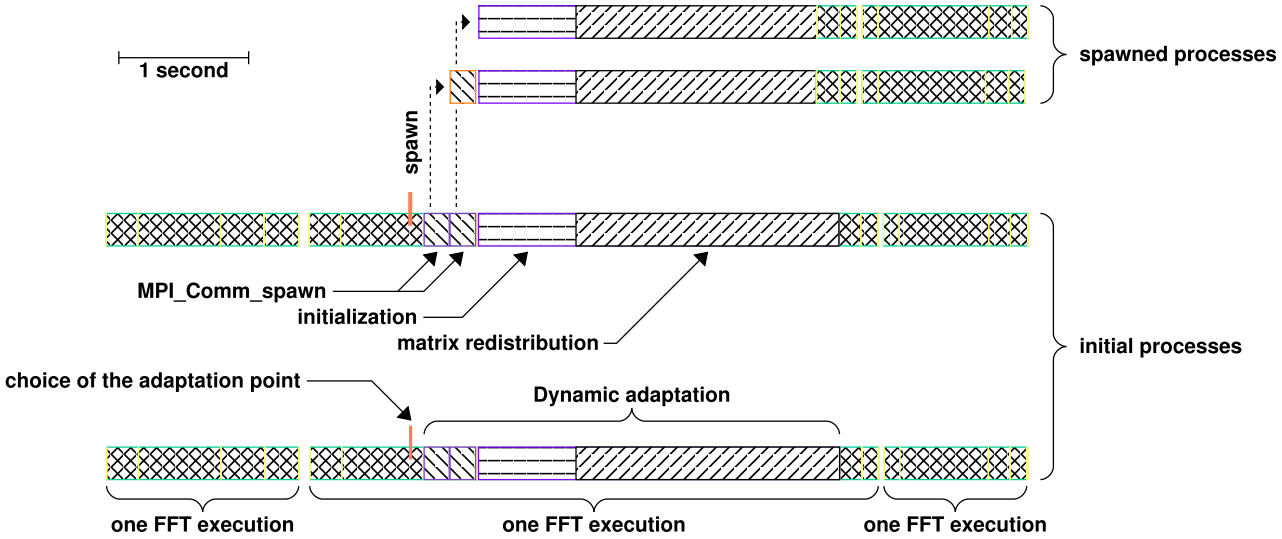


Figure 5. Execution trace of an adaptation that spawns processes

This trace shows that the choice of the adaptation point is done concurrently to the execution of the service. Then, the effective execution of the reaction is postponed to the chosen adaptation point, further in the future of the execution.

The plan begins by spawning new processes. Due to the MPI-2 specification, and in order to be able to stop each process independently of the others, each process has to be spawned individually. In order to simplify the manipulation of MPI communicators, spawned processes participate to the creation of the following processes. This is why one of the spawned processes has a call to MPI_Comm_spawn. Once processes have been spawned, some initialization is performed. This initialization action computes the values that depends on the set of processes, such as communicator objects. Then, matrices are redistributed among the new collection of processes. Endly, the execution of the FFT that was in progress resumes.

5.3. Overhead of the framework

In order to measure the overhead of the proposed framework, the demonstration component has been executed without any adaptation. This experiment permits to evaluate the overhead of the annotations required by the framework. For this experiment, the component executes 2000 iterations on a 16 machines cluster. Table 6 summerizes the execution time of each call to the framework in microseconds. The high maximum value for “function enter” appears to correspond to the first calls for each process. This can be explained by the absence of a complete warmup phase.

function	minimum	mean	maximum	calls per iteration
Adaptation point	14.	21.76	138.	6
Enter condition	7.	10.74	68.	3
Enter function	16.	19.63	510.	1
Enter loop	43.	45.94	58.	0
Fastforward	6.	19.09	104.	7
Iterate in loop	10.	14.05	132.	1
Leave condition	7.	17.34	180.	3
Leave function	8.	9.38	93.	1
Leave loop	9.	13.70	90.	1
Iteration body	777536.	852310.96	1560923.	

Figure 6. Execution times (in microseconds)

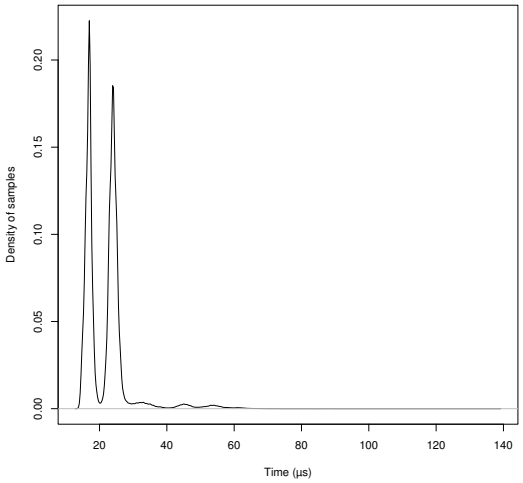


Figure 7. Distribution of measured execution times for the “adaptation point” function

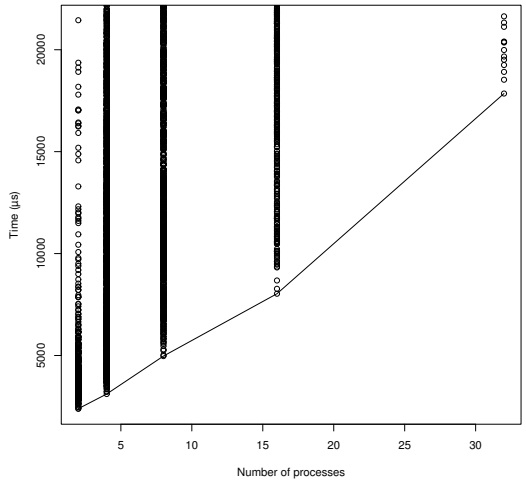


Figure 8. Scalability of the algorithm for choosing the adaptation point

Figure 7 shows the distribution of the measured times amongst the samples for the “adaptation point” function. This curves shows that two execution times have a high frequency: the lowest one corresponds to favorable cache situations; the highest one to defavorable cache situations. The same phenomenon appears with the other functions.

Given the number of calls per iteration for this sample component, the ratio of time lost because of the framework is under 0.05%. Given this result, it appears that the overhead of the framework can be considered as negligible for real world applications.

5.4. Scalability of the choice of the point

As our coordinator solves an agreement problem, the more processes are used, the more time it takes. In order to evaluate the scalability of the algorithm involved in our coordinator, the demonstration component has been executed and adapted with a parallel degree ranging from 2 to 32 processes. Figure 8 shows the time used for choosing the adaptation point at which the reaction is executed.

Care should be taken while interpreting the results. Indeed, the measured time depends not only on the number of processes, but also on the exact time at which the algorithm is triggered (and the

execution time of the service code between adaptation points). Whereas we want to evaluate the former, the latter can not be controlled and scrambles the measures. Computing the minimum time for each number of processes eliminates most of the noise caused by the variation of the trigger time if enough measures are done.

On the figure, dots represent measures; minima for each number of processes are connected by a line. This line appears to evolve almost linearly with regard to the number of processes. This result could be expected as the actual implementation relies on a ring communication scheme.

6. Related works

Several works have proposed architectures for dynamic adaptation such as [1,5,7,8,11,12]. Despite different architectures and approaches, those projects rely on concepts and functionnalities similar to the ones of our approach. Whereas many projects have studied dynamic adaptation in the context of mobile computing, only few are interested in this problem in the area of parallel computing. As the problem described in section 4 of coordinating the execution of the actions appears mainly in the context of parallel computing, many projects did not study it.

Whereas our approach focuses on building adaptable components by extending standard components, the ASSIST [1] approach for dynamic adaptation is based on high-level parallel language constructs. With this approach, the compiler itself is able to emit code for handling dynamicity. Whereas our approach gives full control of dynamic adaptation to the developer, the ASSIST approach permits some dynamic adaptation transparently to the developer. In addition, knowledge of the generated code can be used to specialize the runtime support for dynamic adaptation.

The PCL project [7] aims at easing the construction of adaptable distributed applications. It focuses on how the application can be modified for dynamic adaptation thanks to a runtime providing some reflexive programming support. In order to support reflection, PCL introduces the notion of “adapt sites”, which are special nodes in the control flow graph that contain collections of unordered (and potentially concurrent) tasks. Intercession operators allow modifications of the collection of tasks associated to each adapt site. In addition, PCL defines a language for expressing when and how the application should be adapted thanks to “adapt methods”. Whereas PCL mixes in a single function probes query, decision-making and planning of the adaptation, which may ease the global understanding of the adaptation, our framework separates these concerns in distinct components, which may simplify the design and reuse of more complex adaptation strategies. Endly, PCL defines a model for synchronizing the adaptation [6]. A comparison of the PCL model to ours is given in [3].

7. Future work

Although several projects address the problem of dynamic adaptation, only few of them provide developers with an abstract model of dynamic adaptation. Providing tools to design and reason about dynamically adaptable software is one of the upcoming challenges. A basis for a design methodology has been proposed in [9]. We are currently working with the team producing ASSIST [1], which includes facilities for dynamic adaptation, in order to propose a common abstract model that could be mapped to our frameworks. It could be expected from such a work to provide conceptual tools to ease the design of dynamic adaptation independently of the concrete platform.

Resource disappearance can be dealt with fault-tolerance mechanisms. However, fault-tolerance focuses on sudden resource disappearance, whereas maintenance operations for example could be announced in advance. Such announcements should be used to anticipate resource disappearance instead of blindly waiting for a fault to occur. Moreover, fault-tolerance approaches fail to make the

application benefit from appearing resources. On the other side, the event-based nature of dynamic adaptation makes it particularly suitable when changes are announced in advance. Fault-tolerance and our approach to dynamic adaptation are complementary in their way to address the dynamicity of the execution environment. Convergence of the two approaches within a single framework should be investigated. In particular, the synchronization concern of the coordinator functional box within dynamically adaptable components introduces the notion of "point". This notion can be compared to checkpoints that can be used to implement fault-tolerance. Although the two notions do not match exactly, they might rely on the same infrastructure, possibly leading to a unified framework.

References

[1] Marco Aldinucci, Sonia Campa, Massimo Coppola, Marco Danelutto, Domenico Laforenza, Diego Puppin, Luca Scarponi, Marco Vanneschi, and Corrado Zoccolo. Components for high performance grid programming in the grid.it project. In *Workshop on Component Models andd Systems for Grid Applications*, June 2004.

[2] Jérémy Buisson, Françoise André, and Jean-Louis Pazat. Dynamic adaptation for grid computing. In P.M.A. Sloot, A.G. Hoekstra, T. Priol, A. Reinefeld, and M. Bubak, editors, *Advances in Grid Computing - EGC 2005 (European Grid Conference, Amsterdam, The Netherlands, February 14-16, 2005, Revised Selected Papers)*, volume 3470 of *LNCS*, pages 538–547, Amsterdam, February 2005. Springer-Verlag.

[3] Jérémy Buisson, Françoise André, and Jean-Louis Pazat. Enforcing consistency during the adaptation of a parallel component. In *The 4th International Symposium on Parallel and Distributed Computing*, July 2005.

[4] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[5] Pierre-Charles David and Thomas Ledoux. Towards a framework for self-adaptive component-based applications. In *DAIS'03*, volume 2893 of *LNCS*. Springer-Verlag, November 2003.

[6] Brian Ensink and Vikram Adve. Coordinating adaptations in distributed systems. In *24th International Conference on Distributed Computing Systems*, pages 446–455, March 2004.

[7] Brian Ensink, Joel Stanley, and Vikram Adve. Program control language: a programming language for adaptive distributed applications. *Journal of Parallel and Distributed Computing*, 63(11):1082–1104, November 2003.

[8] John Keeney and Vinny Cahill. Chisel: a policy-driven, context-aware, dynamic adaptation framework. In *4th International Workshop on Policies for Distributed Systems and Networks (POLICY'03)*, pages 3–14. IEEE, 2003.

[9] Malcolm McIlhagga, Ann Light, and Ian Wakeman. Towards a design methodology for adaptive applications. In *Mobile Computing and Networking*, pages 133–144, May 1998.

[10] NAS. Parallel benchmark. http://www.nas.nasa.gov/Software/NPB/.

[11] Pierre-Guillaume Raverdy, Hubert Le Van Gong, and Rodger Lea. DART : a reflective middleware for adaptive applications. In *OOPSLA'98 Workshop #13 : Reflective programming in C++ and Java*, October 1998.

[12] Maria-Teresa Segarra and Françoise André. A framework for dynamic adaptation in wireless environments. In *Technology of Object-Oriented Languages and Systems (TOOLS 33)*, pages 336–347. IEEE, 2000.

Towards a distributed scalable data service for the Grid*

M. Aldinucci[a], M. Danelutto[b], G. Giaccherini[b], M. Torquati[b], and M. Vanneschi[b]

[a]Institute of Information Science and Technologies (ISTI) – National Research Council (CNR),
Via Moruzzi 1, I-56124 Pisa, Italy

[b]Department of Computer Science, University of Pisa,
Largo B. Pontecorvo 3, I-56127 Pisa, Italy

Abstract: ADHOC (Adaptive Distributed Herd of Object Caches) is a Grid-enabled, fast, scalable object repository providing programmers with a general storage module. We present three different software tools based on ADHOC: A parallel cache for Apache, a DSM, and a main-memory parallel file system. We also show that these tools exhibit a considerable performance and speedup both in absolute figures and w.r.t. other software tools exploiting the same features.

Keywords: Grid, Data Grid, Web caching, Apache, PVFS, DSM, Web Services.

1. Introduction

The demand for performance, propelled by both challenging scientific and industrial problems, has been steadily increasing in past decades. In addition, the growing availability of broadband networks has boosted data traffic and therefore the demand for high-performance data servers. Distributed memory Beowulf clusters and Grids are gaining more and more interest as low cost parallel architectures meeting such performance demand. This is especially true for industrial applications that require a very aggressive development and deployment time for both hardware solutions and applications, e.g. software reuse, integration and interoperability of parallel applications with the already developed standard tools.

However, these needs become increasingly difficult to be met with the growing scale of both software and hardware solutions. The Grid is a paradigmatic example. The key idea behind Grid-aware applications consists in making use of the aggregate power of distributed resources, thus benefiting from a computing power that falls far beyond the current availability threshold in a single site. However, developing applications able to exploit it is currently likely to be a hard task. To realize the potential, programmers must design highly concurrent applications that can execute on large-scale platforms that cannot be assumed neither homogeneous, secure, reliable nor centrally managed. Also, these applications should be fed with large distributed collections of data.

ADHOC (Adaptive Distributed Herd of Object Caches), is a distributed *object* repository [3]. It provides applications with a distributed storage manager that virtualizes Processing Elements (PEs) primary or secondary memories into a unique common memory. However, it is not just another Distributed Shared Memory (DSM), it rather implements a more basic facility. The underlying idea of ADHOC design is to provide the application (and programming environment) designer with a toolkit to solve data storage problems in the Grid framework. In particular, it provides the programmer with building blocks to set up client-server and service-oriented infrastructures which can cope with Grid difficult issues aforementioned. The semi-finished nature of ADHOC ensures high adaptability and extendibility to different scenarios, and rapid development of highly efficient storage and buffering

*This work has been supported by the Italian MIUR FIRB *Grid.it* project No. RBNE01KNFP.

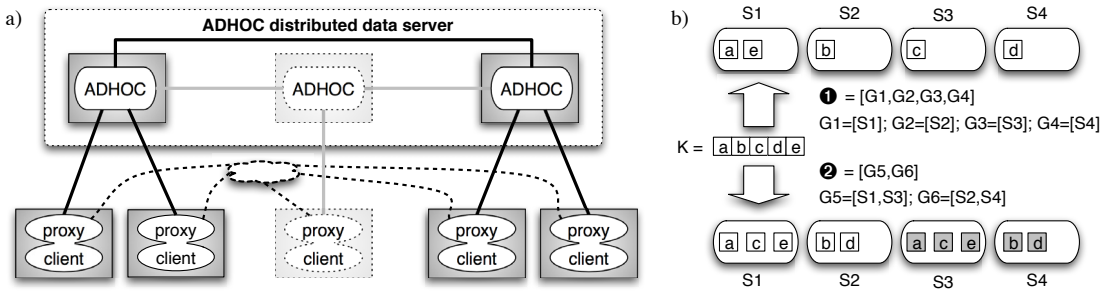


Figure 1. a) Typical architectural schema of applications based on ADHOC. b) Example of two different distribution and replication schemas (❶ and ❷) for a collection K of objects a, b, c, d over 4 ADHOC servers S1, S2, S3, S4 (grey objects are replicas).

solutions meeting industrial needs.

In this paper, we discuss ADHOC and its grid-oriented features. Also, we present the design of three different ADHOC-based software tools and we compare their performance with others exploiting similar features:

1. A cache built on top of ADHOC for farms of the *Apache* Web server. It enables a farm of Apache web servers to exploit the aggregate memory space and network bandwidth of many PEs with a sensible speedup w.r.t. native Apache cache, and with no modification to the Apache core since it can be attached as plug-in.
2. A object based DSM for ASSIST [2], which is a high-level programming environment for Grid applications. ADHOC with a suitable proxy library provides ASSIST with a shared memory abstraction matching typical Grid requirements by supporting heterogeneity and dynamic availability of platforms.
3. ASTFS, a PVFS-like parallel virtual file system. Differently from PVFS1 [5], it supports heterogeneous platforms and data caching, while performing better or comparably w.r.t. PVFS working on a RAM-disk file system.

2. The ADHOC data server

The ADHOC underlying design principle consists in clearly decoupling the management of computation and storage in distributed applications. The development of a parallel/distributed application is often legitimated by the need of processing large bunches of data. Therefore, data storages are required to be fast, dynamically scalable and enough reliable to survive to some hardware/software failures. Decoupling helps in providing a broad class of parallel applications with these features while achieving very good performances. ADHOC virtualizes a PE primary (or secondary) memory, and cooperating with other ADHOCs, it provides a common distributed data repository.

The general ADHOC-based architecture is shown in Fig. 1. Clients may access data through different protocols, which are implemented on client-side within *proxy* libraries. Proxies may act as simple adaptors, or exploit complex behaviors also cooperating with other client-side proxies (e.g. distributed agreement, dotted lines in the figure). Both clients and servers may be dynamically attached and detached during the program run.

A set of ADHOCs implements an *external* storage facility, i.e. a repository for arbitrary length,

contiguous segments of data (namely *objects*). An object cannot be spread across different ADHOCs, it can be rather replicated on them. Objects can be grouped in ordered *collections* of objects, which can be spread across different ADHOCs.

Both objects and their collections are identified by *keys* with fixed length. In particular, the key of a collection specify to which *spread-group* and *replica-group* the collection belong. These groups logically specify how adjacent objects in the collection are mapped and replicated across a number of logical servers. The actual matching between logical servers and ADHOCs is performed at run-time through a distributed hash table. ADHOC API enables to `get/put/remove/execute` an object, and to `create/destroy` a key for a collection of objects. ADHOC does not provide collective operations to manage collections of objects (except key creation and destruction), these collective operations can be implemented within the client proxy. Each ADHOC manages an *object storage* and a write-back *cache* that are used to store server home objects and remote home objects respectively.

An example is shown in Fig. 1 b). Adjacent objects a, b, c, d, e of the collection K are stored in the distributed data server in two different ways (❶ and ❷). Adjacent objects of a collection are allocated and stored in a round robin way along a list of replica-groups. Each object is stored in each server appearing in the replica-group. Many spread-groups and replica-groups can be defined for a distributed data server, moreover they can be dynamically created and modified. This enables both to attach new ADHOCs to a distributed server and to re-map (migrate) objects among different ADHOCs within a distributed server. Once an ADHOC does not appear in any group and is empty, it can be easily detached with no data loss (it can also detached at any moment, possibly with partial data loss). Object re-mapping might be an expensive operation and is supposed to be infrequent. Notice that since the collection and object keys remain unchanged in re-mapping, data may be re-mapped at run-time while keeping valid all involved keys. As an example, a distributed linked list using keys as pointers may be transparently re-mapped.

ADHOC `execute(key,method)` operation enables the remote execution of a method, provided the key refers a chunk of code instead of plain data (i.e. an actual object which is executable on the target platform). This operation is meant as mechanism to extend server core functionalities for specific needs. As an example, lock/unlock, object consistency management, and atomic sequences of operations (e.g. `get&remove`) on objects have been introduced in ADHOC in this way.

As sketched in Fig. 2, ADHOCs can be connected though firewalls and across networks exploiting different private address ranges. In particular:

- ADHOCs can connect one another with a configurable number and range of ports. An ADHOC-based distributed server with n ADHOCs can be set up across n firewalls, $n-1$ of them having outbound connectivity only, and 1 firewall having just 1 open in-bound port. However, the richer is the connectivity among servers the better is the expected performance.
- ADHOCs may work as relays for others. This enable to set up a distributed data server across networks with different private address ranges, that is the usual configuration of clusters belonging to a Grid. For `get/put` objects, each connected graph of ADHOCs is functionally equivalent to a complete graph. However, currently only directly connected ADHOCs may belong to the same spread- or replica-group (collection cannot be spread through relays). Moreover, since ADHOCs may be dynamically attached, different subgraphs are not supposed to be started all together, as may happen in the case they are executed through different job schedulers on top of different clusters.

2.1. ADHOC Implementation

An ADHOC is implemented as a C++ single thread process; it relies on non-blocking I/O to manage concurrent TCP connections [6]. The ADHOC core consists of an executor of a dynamic set of

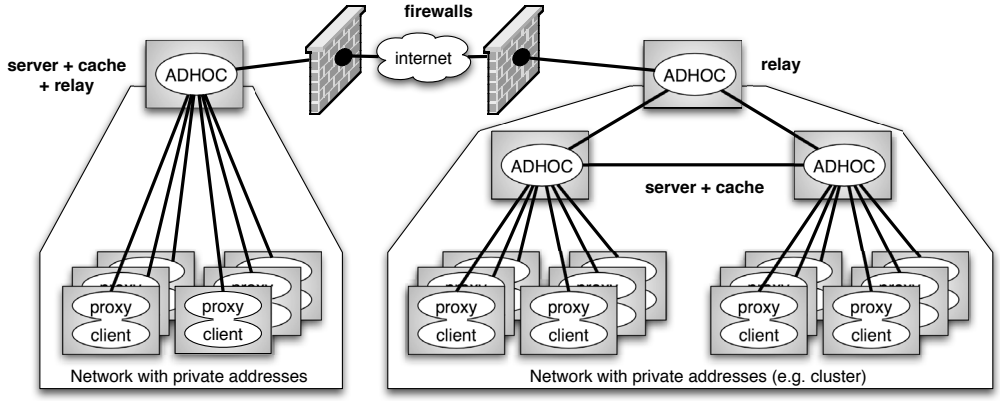


Figure 2. ADHOC-based distributed data server running onto different clusters with private address ranges and protected by firewall.

finite state machines, namely *services*, which reacts to socket-related events raised by O.S. kernel (i.e. connections become writable/readable, new connection arrivals, connection closures, etc.). In the case one service must wait on an event, it consolidates its state and yields the control to another one. The ADHOC core never blocks on I/O network operations: neither on `read()/write()` system calls nor on ADHOC protocol primitives like remote PE memory accesses. The event triggering layer is derived from the `Poller` interface [9], and may be configured to use several POSIX connections multiplexing mechanisms, such as: `select`, `poll`, `kqueue`, and Real-Time signals. Non-blocking I/O on edge-triggered signaling mechanism is known to be the state-of-the-art of server technologies over TCP [7]. Indeed, an ADHOC can efficiently serve on a single port many clients, each of them supporting thousand of concurrent connections.

We experienced that the ADHOC-based distributed data server exhibits a close to perfect speedup in parallel configuration (many connected ADHOCs), both in supplied memory room and aggregate network bandwidth. It also supports heterogeneous distributed platforms, in particular it has been extensively tested on Linux (2.4.x/2.6.x) and Mac OS X (10.3.x/10.4.x). We refer back to [3] for any further detail on ADHOC implementation and testing.

2.2. ADHOC as a Grid-aware software

ADHOC is a part of the ASSIST Grid-aware programming environment [1], and it is building block for Grid-aware applications and programming environments because it can cope with many of the key issues of the Grid:

- *Connectivity:* firewalls, multi-tier networks with private address ranges.
- *Performance and fault-tolerance:* data distribution, replication, and caching (parallelism and locality), dynamic data re-distribution, adaptability through dynamic reconfiguration of the set of machines composing the distributed server.
- *Heterogeneity and deployment:* it is free GPL software that can be easily ported on POSIX platforms; it has been tested on several Linux and BSD platforms; it supports heterogeneous clusters (in O.S. and CPU); it can be deployed through standard middleware (as Globus); several ADHOCs composing a single distributed server do not need to start all together, thus they can be deployed on different clusters through different job schedulers.

Unlike some other approaches to data grid (e.g. European Data Grid [10]) ADHOC does not provide a rigid middleware solution for a particular problems (e.g. very large, mostly read-only scientific data). It rather provides the application developer with a configurable and extendible building block to target quite different problems, in both scientific and industrial computing, ranging from high-throughput grid data storage to low-latency high-concurrency cluster and enterprise grid data services.

3. Apache Web Caching Experiments

The ADHOC+Apache architecture is compliant to Fig. 1 a). In this case the *client* is the Apache Web server, the *proxy* is a modified version of *mod_mem_cache* Apache module and dashed lines are not present. In particular, *mod_mem_cache* has been modified by only substituting local memory allocation, read and write with ADHOC primitives.

Observe that ADHOC+Apache architecture is designed to improve Apache performance whether the performance bottleneck is memory size, typically in the case the working set does not fit the main memory. In all other cases, the ADHOC+Apache architecture does not introduce any significant performance penalties w.r.t. the stand-alone Apache equipped with the native cache.

We measured the performance of Apaches+ADHOC architecture on a 21 PEs RLX Blade; each PE runs Linux (kernel-2.6.x) and is equipped with an Intel P3@800MHz, 1GB RAM, a 4200rpm disk and a 100Mbit/s switched Ethernet devices. The data set is generated according to [4] by using a Zipf-like request distribution ($\alpha = 0.7$), and has a total size of 4GBytes. In all tests we used the Apache 2.0.52 Web server in the *Single-Process Multi-Threaded*. HTTP requests are issued by means of the *httperf* program. In Fig. 3 we compare Apache against Apache+ADHOC performances. The test takes in to account three basic configurations:

① an Apache+ADHOC running on different PEs, ADHOC exploiting 900MB of *object storage* total memory accessed by all Apache threads.
② a stand-alone Apache with no cache.
③ a stand-alone Apache with the *mod_mem_cache* (Apache native cache) exploiting a maximum of 900MB.

As shown by ③, the Apache with the original cache lose its stability when the requests rate grows. In this case, Apache spawn more and more threads to serve the increasing pressure of requests, inducing harmful memory usage: the competition of cache subsystem and the O.S. in both memory space and memory allocation leads the O.S. to the swap border resulting in a huge increase of reply latency. Quite surprisingly, the Apache with no cache performs even better (②). In reality this behavior is due to the File System buffer that acts as a cache for Apache disk accesses, and which gracefully decrease its size in case the system requires more memory to manage many threads avoiding swapping. In this case the performance also depends on site organization on disks. In general, the FS cache is unsuitable for Web objects since requests do not exploit spatial and temporal locality w.r.t. disk-blocks [4]. Moreover, FS cache is totally useless for dynamic Web pages, for which we experienced the effectiveness of the Apache native cache module [8]. As a matter of fact, the 2PEs figures (③) confirm that mapping Apache and ADHOC on different PEs significantly improves performances.

As shown in [3], the gain of the Apache+ADHOC architecture is even greater for Apache Multi-Process Multi-Threaded configuration since Apache processes can share a common memory through ADHOC. Additional experiments on parallel configuration confirm that a single ADHOC may support many Apaches with a very good scalability [3].

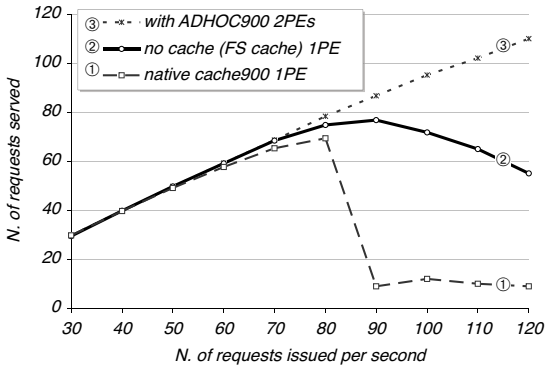


Figure 3. Evaluation of ADHOC as Apache cache.

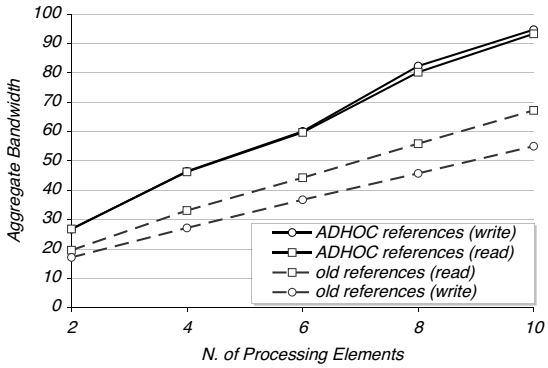


Figure 4. Evaluation of ADHOC-based DSM.

4. An ADHOC-based DSM for ASSIST

ASSIST is a programming environment aimed at the development of distributed high-performance applications on Grids [2]. ASSIST applications are described by means of a coordination language, which can express arbitrary graphs of either sequential or parallel components, which may exploit distributed shared data structures.

Up to now, these data structures were stored in a standard DSM implemented within ASSIST runtime. This DSM was implemented as a library providing typed global pointers (called *references*), it requires the full connectivity with all partners, and can hardly deal with firewalls. The last version of ASSIST (v1.3) includes a novel ADHOC-based implementation of references. This new implementation overcome mentioned problems of the previous version while improving the overall DSM flexibility and performance:

- Data is stored externally to computation in specialized servers. The data can now survive to application lifespan, and it can independently mapped from computational activities. Figure. 4 describes aggregate bandwidth of reading/writing a spread array (`int[800M]`, as a collection of 16KB objects). The ADHOC-based solution exhibit a close to optimal absolute figures, and a clear performance gain with respect the previous version of the DSM.
- ADHOC can be dynamically started and stopped to meet variable application requirements and Grid platforms availability over time.
- Data can be migrated among different ADHOCs to implement dynamic load-balancing schemas for memory allocation and network throughput, meeting Grid platforms unsteadiness and changing load over time.
- ADHOC-based distributed data server can easily wrapped to supply a HTPP/SOAP data service for the Grid.

5. An ADHOC-based Parallel File System

The Parallel Virtual File System (PVFS) [5,11] is one of the most used high-performance and scalable parallel file system for PC clusters that requires no special hardware.

In order to provide high-performance access to data stored on the file system by many clients, PVFS spreads data out across multiple cluster I/O nodes (IONs). By spreading data across multiple

I/O nodes, applications have multiple paths to data through the network and multiple disks on which data is stored. This eliminates single bottlenecks in the I/O path and thus increases the total potential bandwidth for multiple clients, or aggregate bandwidth. Metadata stored in a special node, called manager (MNG). Metadata is information that describes a file, such as its name, its place in the directory hierarchy, its owner, and how it is distributed across nodes in the system. When reading a file from PVFS, a client contacts MNG node to retrieve file meta-data, then it gather file parts, each from the proper IONs. As show in Fig. 5 a), a PVFS client may benefit from the concurrent connection to many IONs, thus benefiting from network aggregate bandwidth.

The ADHOC-based FS (called ASTFS) implements the same functionalities of PVFS, and exploits a similar API. ASTFS API is realized by means of a proxy library linked to the application client (see Fig. 1) which translates file-oriented commands in a sequence of ADHOC commands. Figure 5 c) reports a comparison, in terms of aggregate bandwidth at the client ends, between PVFS and ASTFS (tested architectures are shown in Fig. 5 a) and b), respectively); ASTFS is configured to exploit the full connectivity among ADHOC servers. As shown in Fig. 5 c), PVFS and ASTFS (working on RAM-disk) exhibit quite the same bandwidth, which should be considered a very good result since PVFS represent the state of the art of distributed FS for clusters. ASTFS, differently from PVFS, neither requires the full connectivity among PEs (client-server, server-server). As a matter of fact, grid nodes do not exhibit a complete graph of links/connections due to the multi-tier network structure, firewalls, private address ranges in clusters, etc. We also experienced that a single link between clients is not a limiting factor for client throughput towards the servers. In particular, ASTFS multi-threaded proxy succeeds to pipeline multiple requests on a single link relying on the ability of ADHOC in sustaining a very high number of concurrent requests on it.

ASTFS has been primary designed to manage main memory allocation, but it can be easily configured to use disk as storage. It also exhibits some additional features with respect to PVFS, such as the support for heterogeneous platforms, and for caching of read-only opened files. As shown in Fig. 5 c) the FS performance significantly increases whether each ADHOC is configured to exploit a little cache (10 MB). As typical for caching, the equivalent speedup in terms of aggregate bandwidth is super-linearly boosted.

The same test has been also performed in the Fig. 2 scenario: the two 6PEs clusters are hosted in two different institutes of Pisa University (each of them protected by its own firewall with one open port), they are internally linked with a fast Ethernet, and connected one to the other with a 2 MB/s link (average). Just the front-end machine of each cluster can connect to its counterpart in the other cluster. On this scenario, we experienced an aggregate bandwidth at clients ends of 34 MB/s with no cache. Notice that the ADHOC hosted in the font-end node can be configured to exploit a cache of remote sites data considerably improving the overall performance.

6. Conclusions

Overall, we envision a complex application made up of decoupled components, each delivering a very specific service. Actually, ADHOC provides the programmer with a data sharing service. We introduced ADHOC, a fast and scalable "storage component" which cope with many of Grid key issues, such as performance, adaptivity, fault-tolerance, multi-site deployment and run (despite of firewalls, job schedulers, private addresses). We shown that ADHOC simplicity is its strength: it enabled the rapid design and development of three different memory management tools in few months. These tools exhibit a comparable or better performance with respect their specialized counterparts. In some cases, applications relying on these tools became ready for the Grid with no modifications to their code. ADHOC is freely available under GPL license as part of the ASSIST toolkit.

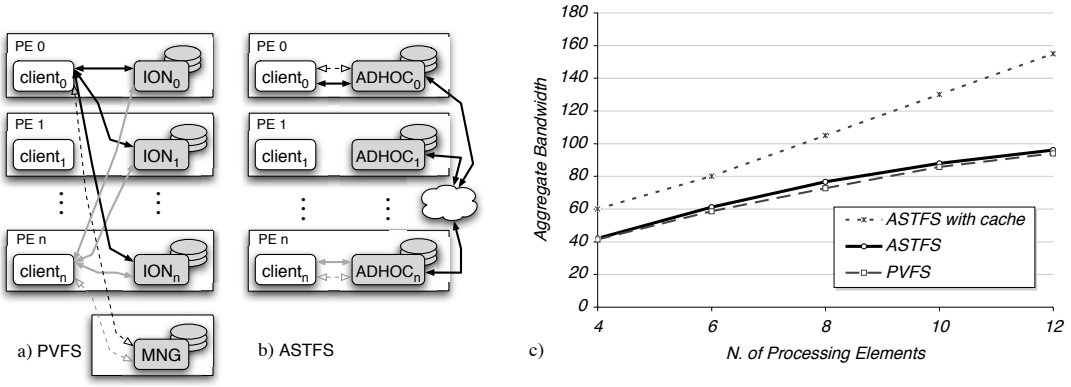


Figure 5. a) Many clients accessing a PVFS. Each PE hosts a client is required to be an ION. Solid edges show data paths, dashed edges meta-data paths. b) Many clients accessing an ASTFS. c) Comparison among PVFS and ASTFS. Each client read a partition of the file in segments of 10MB randomly chosen within its partition. In both cases, each client reads the same 800MByte file spread among all PEs.

References

[1] M. Aldinucci, S. Campa, M. Coppola, M. Danelutto, D. Laforenza, D. Puppin, L. Scarponi, M. Vanneschi, and C. Zoccolo. Components for high performance Grid programming in Grid.it. In V. Getov and T. Kielmann, editors, *Proc. of the Workshop on Component Models and Systems for Grid Applications*, CoreGRID series. Springer Verlag, January 2005.

[2] M. Aldinucci, M. Coppola, M. Danelutto, M. Vanneschi, and C. Zoccolo. ASSIST as a research framework for high-performance Grid programming environments. In J. C. Cunha and O. F. Rana, editors, *Grid Computing: Software environments and Tools*. Springer Verlag, January 2006.

[3] M. Aldinucci and M. Torquati. Accelerating apache farms through ad-HOC distributed scalable object repository. In M. Danelutto, M. Vanneschi, and D. Laforenza, editors, *10th Intl Euro-Par 2004: Parallel and Distributed Computing*, volume 3149 of *LNCS*, pages 596–605, Pisa, Italy, August 2004. Springer Verlag.

[4] L. Brelau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *Proc. of the Infocom Conference*, 1999.

[5] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur. PVFS: A parallel file system for linux clusters. In *Proc. of the 4th Linux Showcase and Conference*, pages 317–327, Atlanta, GA, USA, October 2000.

[6] A. Chandra and D. Mosberger. Scalability of Linux event-dispatch mechanisms. Technical Report HPL-2000-174, HP Labs., Palo Alto, USA, December 2000.

[7] L. Gammo, T. Brecht, A. Shukla, and D. Pariag. Comparing and evaluating epoll, select, and poll event mechanisms. In *Proc. of the Ottawa Linux Symposium*, Ottawa, Canada, 2004.

[8] A. Iyengar and J. Challenger. Improving Web server performance by caching dynamic data. In *Proc. of the USENIX Symp. on Internet Technologies and Systems Proceedings*, Berkeley, CA, USA, Dec. 1997.

[9] D. Kegel. *Poller Interface*, 2003. (`http://www.kegel.com/poller/`).

[10] E. Laure, H. Stockinger, and K. Stockinger. Performance engineering in data Grids. *Concurrency and Computation: Practice and Experience*, 17(2–4):171–191, 2005.

[11] *The PVFS home page*. (`http://www.parl.clemson.edu/pvfs/index.html`).

eNANOS: Coordinated Scheduling in Grid Environments*

I. Rodero[a], F. Guim[a], J. Corbalán[b], J. Labarta[b]

[a]Barcelona Supercomputing Center, Jordi Girona 31, 08034 Barcelona, Spain

[b]Universitat Politècnica de Catalunya, Jordi Girona 1-3, 08034 Barcelona, Spain

One of the current challenges in Grid computing is the efficient scheduling of HPC applications. In the eNANOS project we propose a 3-layer coordinated scheduling architecture for the execution of HPC applications, from the Grid level to the processor scheduling level. In this paper we propose an architecture that allows that the resource broker schedules in coordination with the local resource manager and its local processor scheduler. We are interested in enabling the broker to obtain information about the dynamic behaviour of applications that are running on the local computational resources. Since the Grid technology area is very wide, this project is targeted to High Performance Applications (OpenMP, MPI and MPI+OpenMP) executed on a Grid composed of parallel machines, shared-memory architectures (SMP and CC-NUMA) with a medium to high number of processors. We also show some preliminary results obtained from real workloads which demonstrate the advantages of our coordinated execution environment.

1. Introduction

Grid Computing has emerged in recent years providing a way to perform parallel computing in distributed computational resources. Furthermore, the Grid allows executing jobs in different administrative domains and sharing their existent HPC (High Performance Computing) resources. In order to perform job scheduling and resource management at Grid level, usually it is used a meta-scheduler or a Resource Broker. Typically, the Resource Broker is on top of the Grid infrastructure and it tries to respect the local computational resources policies of each administrative domain and its autonomy. Therefore, the HPC centres usually have local management systems. So, in the execution of Grid applications the meta-scheduler or Resource Broker loses the control of them. We want to take the scheduling decisions being aware of the information reached on lower levels. We think this is the way to achieve an effective scheduling using efficiently the Grid resources. Despite the attempts to define a Grid scheduling architecture [20], that topic is not still fixed and there is no standard available. Even so, it seems reasonable to think that these scheduling layers should interact to each other.

We propose a coordinated scheduling of every component involved in the execution of HPC Grid applications, from the Grid level to the processor scheduling level. We not only want to coordinate the Grid scheduler with the local resource management systems, we also want to perform the scheduling being fully aware of every level which is involved. In particular we are interested in enabling the broker to obtain information about the dynamic behavior of applications that are running on the local computational resources. We consider three basic scheduling levels, from the heterogeneous and dynamic of a Grid to the efficient execution of processes and threads in one or more CPUs of a computer or cluster. This work is part of the eNANOS project [3] whose main aim is the autonomic resource management, one of the required tasks of autonomic computing [11]. Since the Grid

*This research has been supported by the Spanish Ministry of Science and Technology under contract TIN2004-07739-C02-01, and the European Union project HPC-Europa under contract 506079.

technology area is very wide, this project is targeted to High Performance Applications (OpenMP, MPI and MPI+OpenMP) executed on a Grid composed of parallel machines, shared-memory architectures (SMP and CC-NUMA) with a medium to high number of processors.

To carry out an efficient scheduling and for achieve a coordinated scheduling it is also very important the monitoring of both resources and applications. In the eNANOS project the idea is monitoring the jobs in every scheduling level but especially the at the two lowest levels where we want to provide more specific information about the applications behaviour, for instance the reached speedup. This kind of dynamic information can be useful to take decisions about scheduling, as can be dispatching more applications or migrating certain processes or threads. This information can also be very helpful for the researcher because it allows us to see the workload behaviour and precise information in order to improve and adjust the scheduling policies. For this reason we have implemented the NANOS Job Monitor as is shown in section 3.5.

Currently we have implemented the basic mechanisms that coordinate the different layers of the infrastructure and we are working in adding advanced functionalities. Although the complete execution environment is more complex, this paper is centered in the scheduling issues.

2. Related Work

To the best of our knowledge, any developed systems have yet been developed implementing coordinated scheduling between Grid level and other lower levels managing MPI+OpenMP applications as we propose in this paper. An emerging research group at GGF is working in the Grid scheduling architecture and it is introducing some concepts of coordinated scheduling but it is only a preliminary work. There are different initiatives that deal with the problem of scheduling in the three levels that are proposed in this paper. On one hand, we can meet some problems of meta-schedulers or resource brokers for Grid as can be: AppLes [1], Condor-G [7], EZ-Grid [6], GridBus [15], GRMS [9], or GridWay [10]. These brokers implement several policies, for instance policies based on economic models, based on the monitoring information of the resources, or based on prediction mechanisms. On the other hand there are some local job schedulers which are suitable for HPC applications and they work externally from the queuing system, for example MAUI scheduler [13], EASY [19] or OAR Scheduler [17]. They implement several scheduling policies and support some mechanisms as the advanced reservations. There are other projects that are currently implementing systems that include both queuing system and scheduler, and that also advanced reservation mechanisms, for instance, the OAR scheduler, but they are difficult to extend for achieve our requirements: incorporate mechanisms to interact between the CPU and Grid scheduling levels in HPC environments.

3. System Architecture

3.1. General Overview

The system architecture is presented in Figure 1. The Grid Jobs are managed by the eNANOS Resource Broker and submitted to the local HPC resources through the Globus infrastructure. We use LoadLeveler as a queuing system and the NANOS Scheduler as an external scheduler to manage the local jobs. The CPU scheduling is performed by the NANOS-RM and the NANOS Job Monitor provides the monitoring information about the execution of workloads. The information system can be seen as a meta-information system that can collect a very large kind of different information, providing a uniform access to it. The Figure 1 also shows the information flow between the main components of the system architecture. This information is needed to perform a coordinated scheduling.

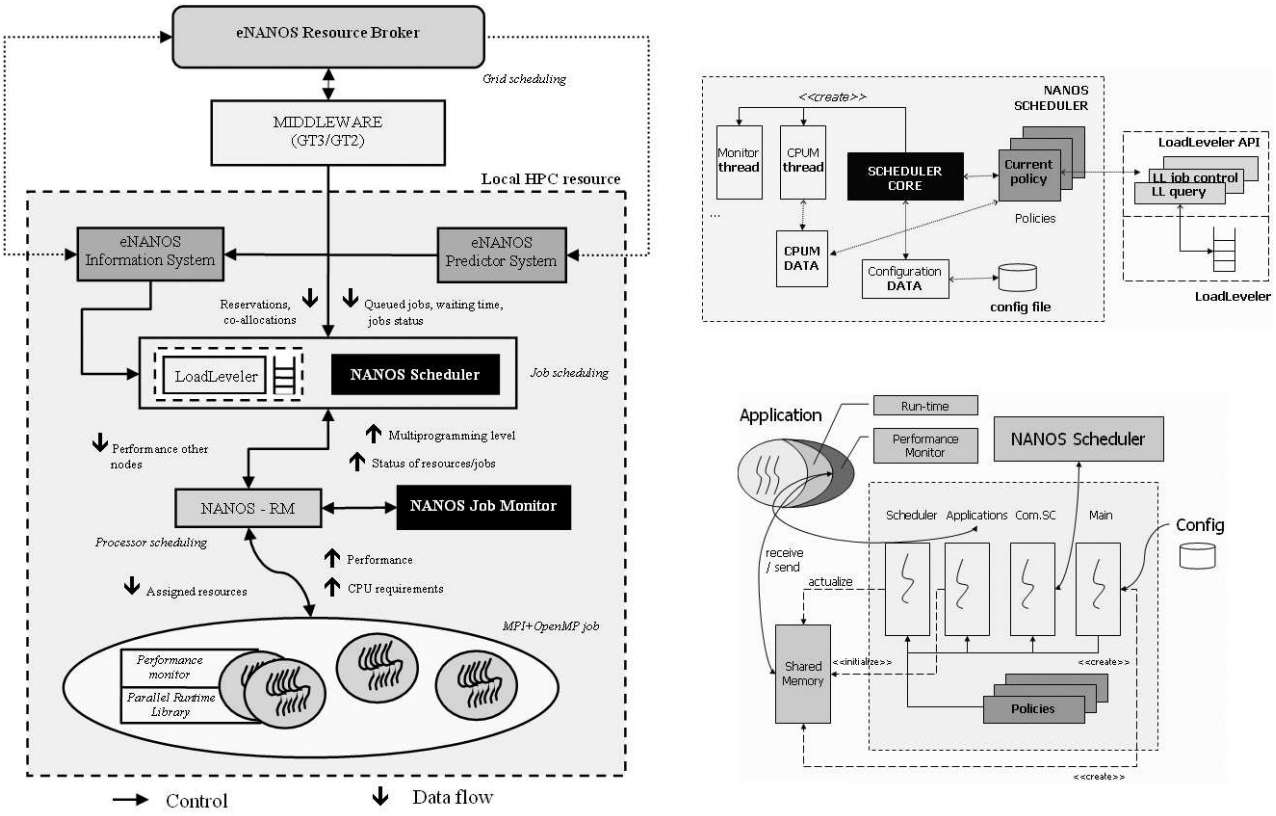


Figure 1. System Overall Architecture (left), NANOS Scheduler Architecture (right-up), NANOS-RM Architecture (right-down)

3.2. eNANOS Resource Broker

The eNANOS Resource Broker is developed on Globus Toolkit 3 as a Grid Service and it is compatible with both GT2 and GT3 services, and implement flexible mechanisms that allow it to become compatible with next Globus versions. It implements the Resource Discovery, Selection and Monitoring, the Job Submission, and the Job Monitoring. Furthermore, the eNANOS Broker provides a set of Grid Service interfaces and a Java API that can be used from command-line clients, applications or portals. Currently the eNANOS broker interacts with our execution environment through the GRAM service provided by the Globus middleware. The broker should obtain information from the Information System and it is also planned to implement scheduling policies based on the information provided by the Predictor Service. More detailed information about this broker can be found in [18].

3.3. NANOS Scheduler

The NANOS Scheduler is responsible to submit and control the jobs which are queued in the queuing system using the LoadLeveler API. The overall architecture of the scheduler is shown in the Figure 1. This scheduler communicates with the NANOS-RM to obtain information related to the applications and the local resources behaviour, and also related to the eNANOS Broker in order to provide information to the upper level. Additionally the scheduler is planned to be communicated to both a prediction service and an information system that are in development. As usual, the scheduler has an iterative behaviour; in each iteration the scheduler performs the following basic functions: searches jobs from the queuing system, updates the system information, evaluates the scheduling

policy, and dispatch the selected jobs

The communication with the other components of the architecture is done through 2 additional threads. The first one is in charge of the interaction with the NANOS-RM. This thread receives information from the runtime about the allowed multiprogramming level by the computational nodes and, also some information related to the applications and hardware, for instance the load of the nodes or the CPUs used by each OpenMP thread. The second one sends the job identifications to the NANOS Job Monitor. It is very important because the Scheduler centralizes the jobs identifications of each layer and is able to map this IDs with the PIDs of processes and threads. It is planned another thread to notify to the Grid level the relevant changes related with the job execution or with the resources. It also can receive instructions from the resource broker but currently, since this functionality is under development, the communication between these components is performed indirectly through the Globus jobmanager and environment variables.

At this moment the scheduler implements the next scheduling policies: FIFO, backfilling and CPUM-based. The policy based on backfilling is implemented with the execution time limits provided by the user. The CPUM-based policy uses the information provided by the NANOS-RM to determine when a node allows the execution of more applications. The runtime information and the multiprogramming level allow us to schedule the next application of the queue following a FIFO policy (depending of the submission time of the job). We are working in a backfilling policy implementation based on the prediction provided for an external prediction service.

3.4. NANOS Resource Manager

The NANOS Resource Manager (NANOS-RM) is a local processor scheduler. Its main goal is to efficiently distribute a set of processors between a set of applications that are under its control. The NANOS-RM offers an API to coordinate with the different components of the system, it interacts with: the parallel applications, the queuing system, the monitoring system, and the performance analysis system. It also includes a set of predefined processor scheduling policies, currently based on space-sharing approaches, to distribute processors. The required functionality to enforce the processor distribution is shared between NANOS-RM and the applications. The NANOS-RM collects the applications requirements, applies the scheduling policy and sends the processor distribution to the applications. We assume that parallel applications are executed in the context of some runtime (OpenMP, MPI) that can provide it the functionality to automatically manage their parallelism. The idea is to modify this runtime to include calls to the NANOS-RM API at the needed points to adjust the processor allocation to the NANOS-RM decisions. The NANOS-RM works in an iterative way (asynchronously) and performs the processors scheduling, the system load control, and the dynamic detection of multilevel applications.

In the Figure 1 is shown the NANOS-RM architecture and the interaction with the NANOS Scheduler. The scheduling policies supported by the NANOS-RM are based on dynamic allocation and have two phases (multilevel). The first phase is between applications, it implements a FIFO policy: one application has N MPI processes and requires a minimum number of N processors. The second phase is between processes of a MPI+OpenMP application and implements two possible policies: Equipartition [12] and Dynamic Processor Balancing [2]. Equipartition starts from the number of cpus allocated to the application and distributes equally among processes of the application. Dynamic Processor Balancing tries to balance the load between the MPI processes of an application. The NANOS-RM interacts with the NANOS Scheduler. It indicates the maximum number of jobs (or multiprogramming level) that the node allows to work without overload problems. The idea is sharing the required information for improving the whole system performance without penalizing the applications performance independently. For the performance analysis, the NANOS-RM detects

the iterative structure of the applications. The spend time to the execution and to the MPI management is measured internally. In order to avoid peaks of unbalance some measures are done and the appropriate iterations are discharged to avoid outliers. More information about the NANOS-RM and the load balancing techniques can be found in [2].

3.5. NANOS Job Monitor

The NANOS Job Monitor collects information about the job execution from both the NANOS-RM and the NANOS Scheduler. It also provides files in XML format describing the execution of the workloads in a computational resource. These XML files follow an schema which contains job information such as the workload name, the Grid job identification (JobID assigned by the eNANOS Broker), the local job identification (LoadLeveler StepID), the job name, the application topology (number of MPI processes and OpenMP threads), and a set of events.

The events describe the life cycle of the threads involved in the job execution. A process event is composed of: the timestamp, the process ID (pid), the thread ID (tid), the status (IDLE, RUNNING, STOPPED, etc.), and CPU (only if the thread status is RUNNING). It also include some events related to the LL actions. The mapping of the job identifications of each layer is very important to the monitoring system because it manages identifications from the grid layer to the process/thread layer. Additionally, we also have implemented a tool to analyze and visualize the workloads. This tool transforms the XML files of the job monitoring to a trace which can be analyzed with the visualization tool Paraver [4]. Another alternative to analyze the workloads previously used in our execution environment is using the IBM Aixtrace tool. With this tool we are able to obtain a binary trace with the system information during an interval of time. With the aixtrace2prv [4] tool the binary trace can be translated to the Paraver format to visualize and analyze it.

3.6. eNANOS Information System

Our system is composed by several entities that provide different kind of information, from the NANOS-RM that can provide information about the state of a process that is running on the system, till the predictor system that provides predictions about the jobs that would be executed in the future. The information system is intended to be a central point of access to all the information related to the system, providing a uniform way to access to the information. It abstracts to the client to which information system the final query is done, it just has to specify which information requires. The information system is not only intended to provide the explained information, it also is intended to provide ways to discover which kind of information is available and how a client can query it, basically providing the XML Schema that describes the format of the XML query. Something important that guided the designing of the IS was that, if it would be necessary, we wanted to add new information systems in an easy way, for instance providing a way to develop modules that could be added to the IS. We could say that this architecture is pretty similar to those architectures presented on monitoring systems as [8], [14], [16], however our information system has to seen as a meta-information system that can collect a very large kind of different information, providing a uniform access to it. Actually we have done a preliminary design of this information system, based on the experience that we have gained on a first stage that consisted on studying the applications information requirements and the characteristics of the some of the most relevant monitoring and information systems that have been published during these lasts recent years [8], [14], [16]. The current state of the work consist on a final design an implementation of the system.

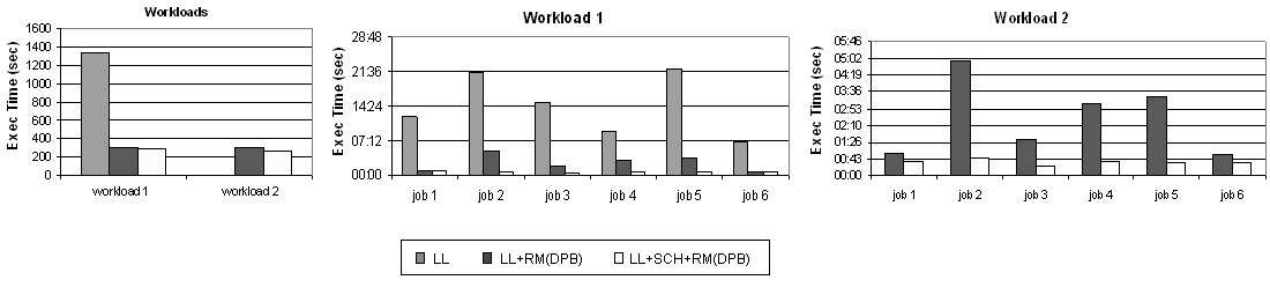


Figure 2. Execution time of the workloads (left) and the individual applications of the workloads

3.7. eNANOS Predictor Service

As has been presented on the previous section this service will be accessed though the information system. This service will provide predictions about the job performance that can be used by both the NANOS Scheduler and the eNANOS broker. It will have a set of prediction techniques, almost all of them will be based in prediction with historical information. We have implemented basically two kinds of prediction techniques: those that are based on statistical approaches that use estimators such as the mean, standard deviation, median and dispersion; and those that are based on data mining algorithms. All the predictors have the similar inputs, such as, user name, group name, number of processes that the job requires, job name and script name. The actual output is a prediction about the user time, system time and memory usage and an interval of confidence that tries to provide information about how accurate can be the prediction. The predicted values can be continuous or discrete; this can be constrained by the query. The prediction service will work basically as a hybrid predictor, depending of the job characteristics and some other input parameters, such as the user, group or system status, it will apply the prediction technique that fits better to it.

4. Preliminary Evaluation

In this paper we present our approach in coordinated scheduling and we are going to evaluate the two lower levels of the architecture. We have evaluated some MPI+OpenMP applications and workloads composed of them in order to show the benefits of this programming model in our execution environment. The evaluation has been performed in an IBM system RS-6000 SP with 8 nodes of 16 Nighthawk Power3 @375Mhz (192 Gflops/s) with 64 Gb RAM and 1.8TB of Hard Disk. The operating system is AIX 5.1 with IBM LoadLeveler queuing system. To evaluate our execution environment we have used the following MPI+OpenMP applications: the NAS Multi Zone (MZ) benchmarks BT, SP of class A, and the CPMD application . We are not going to evaluate scheduling policies; we want to show the potential of our proposed coordinated architecture. To demonstrate the benefits of the architecture we have executed two workloads with the following three configurations:

LL: is the default configuration of the IBM system in which is used a backfilling policy to schedule the queued jobs, there is not any coordination

LL+RM(DPB): the workload is managed by the IBM backfilling scheduler, but the applications are managed by the NANOS-RM with the DPB policy in order to avoid the overloading of the CPUs

LL+SCH+RM(DPB): as well as balancing the CPU load by the NANOS-RM, the job scheduling is done by the NANOS Scheduler in coordination with the NANOS-RM

The experiments are synthetic and are the configuration of them is shown in Table 1, with the

	job1	job2	job3	job4	job5	job6
workload1	BT.A 4x4	BT.A 2x8	SP.A 8x4	BT.A 4x4	SP.A 2x4	SP.A 2x4
workload2	BT.A 4x16	BT.A 2x16	SP.A 8x16	BT.A 4x16	SP.A 2x16	SP.A 2x16

Table 1. Composition of the workloads

application name and the number of MPI processes and OpenMP threads for each application.

In the Figure 2 the execution time of the workloads is shown for the three different configurations. In the first workload with the LL configuration we obtain an execution time higher than with the other two configurations (more or less six times higher). This is caused by an inefficient use of the CPUs. Then, overloading the CPUs of the node can cause undesired situations such as a high number of context switches between processes fighting for CPUs in a short time. In the other two configurations the workload execution time is lower and quite similar between them. With the load balancing in the applications we obtain quiet good results, but in coordination with the local scheduler the results are even better. These improvements are obtained through an efficient scheduling of the applications. We are able to avoid the CPUs overloading by the coordination between the scheduler and the NANOS-RM runtime. We also have to take into account that the LoadLeveler system do not have support for MPI+OpenMP applications, so the backfilling scheduler does not control the second level of parallelism and the taken decisions can be not enough effective. In the second workload every application has the same number of OpenMP threads as the number of CPUs of the node. Thus, we overload the node in a short time during the workload execution, and the second level of parallelism is exploited. Although the number of OpenMP threads is much higher in this workload, the execution time with load balancing is quite similar than the previous case. A good scheduling of the OpenMP level is reflected in a suitable behavior. In this case the improvement in the execution time is better when using the local scheduler and the NANOS-RM together. Due to the workload control is managed by the local scheduler the CPUs are not overloaded.

The individual execution time of the applications that compose each workload is shown in the Figure 2. We can see how in both workloads the execution time of applications is lower when using the Dynamic Processor Balancing. Moreover, the execution time is even much lower when we use the schedulers with coordination. Therefore, with the coordinated scheduling in the local execution environment improve the execution time of the workloads, the throughput, and the response time of the applications. As well as the execution time of the workloads, it is also important the execution time of the applications individually because they are consuming resources during their execution. With the CMPD application we have obtained similar results but with a higher execution time (more than an hour per workload). These two kinds of applications are not problematic because they do not need an intensive use of the resources, the simultaneous execution of several applications do not overload the system. With other applications the load control would be much more important.

5. Conclusions and Future Work

In this paper we have presented our approach in coordinated scheduling of Grid jobs and we have described the main characteristics of the components involved in the eNANOS execution environment. We have discussed that the coordinated scheduling is needed to perform an efficient job scheduling on Grids composed of HPC resources. Furthermore we have presented the evaluation of the lower levels of the scheduling architecture with real workloads. We have seen how the MPI+OpenMP programming model can be a very good choice for the parallel HPC applications

execution on Grids. In the workload evaluation we have seen that the coordinated scheduling between the NANOS Scheduler and the NANOS-RM improve the execution time of the workloads, the throughput and the response time of the individual applications. The system behavior is quite better with the eNANOS execution environment.

As future work our main trend is finishing the eNANOS coordinated scheduling architecture. In general we need to finish and improve all the components of the presented architecture. For instance, at this moment we are designing a more generic and powerful job control interface for the NANOS-RM, to give the job scheduler clients the opportunity to take decisions such as stop running jobs or reduce the allocation of a submitted job. We are also working in the implementation of local scheduling policies for heterogeneous nodes in the NANOS Scheduler. These new policies could be exported to the Grid level taking into account the information of the lower levels. We also have planned to implement new scheduling policies based on prediction techniques (at local scheduler and Grid broker). Moreover, we have to implement the communication between the NANOS-RM and the eNANOS Broker through an information system which is under development. Some other new communication channels should be included or improved. Finally, with our coordinated execution environment it should be easier to implement in the future new functionalities such as checkpointing, coallocation or migration between resources.

References

[1] F. Berman, R. Wolski, S. Figueira, J. Schopf, G. Shao: Application-Level Scheduling on Distributed Heterogeneous Networks. Proceedings of the SC96. 1996

[2] J. Corbalán, A. Duran, J. Labarta: Dynamic Load Balancing of MPI+OpenMP applications. ICPP04, Montreal, Quebec. Canada. 2004

[3] J. Corbalán, R.M. Badia, J. Labarta: eNANOS Performance Tools for Autonomic Grid Resource Allocation Management. CEPBA-IBM Research Institute. 2002

[4] CEPBA Tools Web Site. http://www.cepba.upc.edu/tools_i.htm

[5] CPMD consortium Web Site. http://www.cpmd.org/

[6] EZ-Grid Resource Brokerage System Web Site. http://www.cs.uh.edu/ ezgrid/

[7] J. Frey, T. Tannenbaum, M. Livny, I. Foster, S. Tuecke: Condor-G: A Computation Management Agent for Multi-Institutional Grids. 10th HPDC, IEEE Press. August 2001

[8] Globus Monitoring and Discovery Service (MDS) Web Site. http://www-unix.globus.org/toolkit/mds/

[9] GridLab, A Grid Application Toolkit and Testbed. http://www.gridlab.org

[10] GridWay Project Web Site. http://www.gridway.org

[11] IBM Research, Autonomic Computing Web Site. http://www.research.ibm.com/autonomic/

[12] C. McCan, R. Vaswani, J. Zahorjan: A Dynamic Processor Allocation Policy for Multiprogrammed Shared-Memory Multiprocessors. ACM Trans. on Computer Systems. 1993

[13] MAUI Cluster Scheduler Web Site. http://www.clusterresources.com/products/maui

[14] Mercury Grid Monitoring System Web Site. http://www.lpds.sztaki.hu/mercury/

[15] K. Nadiminti, S. Venugopal, H. Gibbins, R. Buyya: The Gridbus Broker Manual (v.2.0). Grid Computing and Distributed Systems (GRIDS). http://www.gridbus.org/broker

[16] Network Weather Service (NWS) Web Site. http://nws.cs.ucsb.edu/

[17] OAR scheduler Web Site. http://oar.imag.fr/

[18] I. Rodero, J. Corbalán, R.M. Badia, J. Labarta: eNANOS Grid Resource Broker. P.M.A. Sloot et al. (Eds.): EGC 2005, LNCS 3470, Amsterdam. February 2005

[19] J. Skovira, W. Chan, H. Zhon: The EASY - LoadLeveler API Project. LNCS 1162. 1996

[20] R. Yahyapour, P. Wieder: Grid Scheduling Architecture-Requirements. GGF GSA Research Group. January 2005

Parallel program/component adaptivity management

M. Aldinucci[a], F. André[b], J. Buisson[b], S. Campa[c], M. Coppola[a], M. Danelutto[c], C. Zoccolo[c]

[a]Inst. of Information Science and Technologies (ISTI – CNR), V. Moruzzi 1, 56124 Pisa, Italy

[b]University of Rennes I, Avenue du General Leclerc, 35042 Rennes, France

[c]Dept. of Computer Science, University of Pisa, Largo B. Pontecorvo 3, 56127 Pisa, Italy

Grid computing platforms require to handle dynamic behaviour of computing resources within complex parallel applications. We introduce a formalization of adaptive behaviour that separates the abstract model of the application from the implementation design. We exemplify the abstract adaptation schema on two applications, and we show how two quite different approaches to adaptivity, the ASSIST environment and the AFPAC framework, easily map to this common schema.

1. An Abstract Schema for Adaptation

With the advent of more and more complex and dynamic distributed architectures, such as Computational Grids, growing attention has to be paid to the effects of dynamicity on running programs. Even assuming a perfect initial mapping of an application over the computing resources, choices made can be impaired by many factors: load of the used machines and network available bandwidth may vary, nodes can disappear due to network problems, user requirements may change.

To properly handle all these situations, as well as the implicitly dynamic behaviour of several algorithms, *adaptivity* management code has to be built into the parallel/distributed application. In so doing, a trade-off must be settled between the complexity of adding dynamicity-handling code to the application and the gain in efficiency we obtain.

The need to handle adaptivity has been already addressed in several projects (AppLeS [5], GrADS [10], PCL [8], ProActive [4]). These works focus on several aspects of reconfiguration, e.g. adaptation techniques (GrADS, PCL, ProActive), strategies to decide reconfigurations (GrADS), and how to modify the application configuration to optimize the running application (AppLes, GrADS, PCL). In these projects concrete problems posed by adaptivity have been faced, but little investigation has been done on common abstractions and methodology [9].

In this work we discuss, at a very high level of abstraction, a general model of the activities we need to perform to handle adaptivity in parallel and distributed programs.

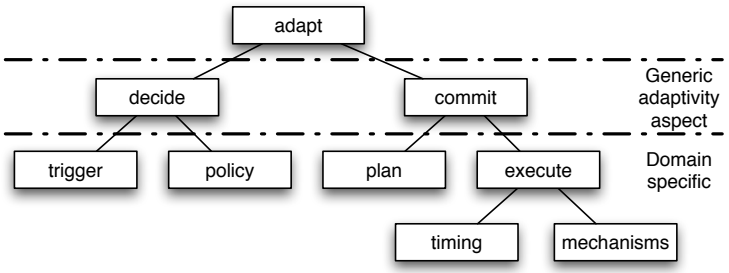


Figure 1: Abstract schema of an adaptation manager.

Our model is abstract with respect to the implemented adaptation techniques, monitoring infrastructure and reconfiguration strategy; in this way we can uncover the common aspects that have to be addressed when developing a programming framework for reconfigurable applications, and we show that it can be applied to two concrete examples: ASSIST [3] and AFPAC [6].

The abstract model of dynamicity management we propose is shown in Fig. 1, where high-level actions rely on lower-level actions and mechanisms. The model is based on the separation of application-oriented abstractions and implementation mechanisms, and is deliberately specified in minimal way, in order not to introduce details that may constrain possible implementations. As an example, the schema does not impose a strict time ordering among its leaves. In order to validate the proposed abstraction, we exemplify its application in two distinct, significant case studies: message-passing SPMD programs, and component-based, high-level parallel programs. In both cases, adaptive behaviour is derived by specializing the abstract model introduced here. We get significant results on the performance side, thus showing that the model maps to worthwhile and effective implementations [3].

The work is structured as follows. Sec. 2 introduces the abstract model. The various phases required by the general schema are detailed with examples in Sec. 3.1 and Sec. 3.2 with respect to two example applications. Sec. 4 explains how the schema is mapped in the AFPAC framework, where self-adapting code is obtained by semi automated restructuring of existing code. Sec. 5 describes how the same schema is employed in the ASSIST programming environment, exploiting explicit program structure to automatically generate autonomic dynamicity-handling code.

2. Adaptivity

The process of adapting the behaviour of a parallel/distributed application to the dynamic features of the target architecture is built of two distinct phases: a decision phase, and a commit phase, as outlined in Fig. 1. The outcome of the decide phase is an abstract adaptation strategy that the commit phase has to implement. We separate the decisions on the strategy to be used to adapt the application behaviour from the way this strategy is actually performed. The decide phase thus represents an abstraction related to the application structure and behaviour, while commit phase concerns the abstraction of the run-time support needed to adapt. Both phases are split into different items. The decide phase is composed of:

- trigger – It is essentially an interface towards the external world, assessing the need to perform corrective actions. Triggering events can result from various monitoring activities of the platform, from the user requesting a dynamic change at run-time, or from the application itself reacting to some kind of algorithm-related load unbalance.
- policy – It is the part of the decision process where it is chosen how to deal with the triggering event. The aim of the adaptation policy is to find out what behavioural changes are needed, if any, based on the knowledge of the application structure and of its issues. Policies can also differ in the objectives they pursue, e.g. increasing performance, accuracy, fault tolerance, and thus in the triggering events they choose to react to.
  Basic examples of policy are “increase parallelism degree if the application is too slow”, or “reduce parallelism to save resources”. Choosing when to re-balance the load of different parts of the application by redistributing data is a more significant and less obvious policy.

In order to provide the decide phase with a policy, we must identify in the code a pattern of parallel computation, and evaluate possible strategies to improve/adapt the pattern features to the current target architecture. This will result in either specifying a user-defined policy or picking one from a library of policies for common computation patterns. Ideally, the adaptation policy should depend on the chosen pattern and not on its implementation details.

In the commit phase, the decision previously taken is implemented. In order to do that, some assessed plan of execution has to be adopted.

- plan – It states how the decision can be actually implemented, i.e. what list of steps has to be performed to come to the new configuration of the running application, and according to which control flow (total or partial order).
- execute – Once the detailed plan has been devised, the execute phase takes it in charge relying on two kinds of functionalities of the support code
  - the different mechanisms provided by the underlying target architecture, and
  - a timing functionality to activate the elementary steps in the plan, taking into account their control flow and the needed synchronizations among processes/threads in the application.

The actual adapting action depends on both the way the application has been implemented (e.g. message passing or shared memory) and the mechanisms provided by the target architecture to interact with the running application (e.g. adding and removing processes to the application, moving data between processing nodes and so on).

The general schema does not constrain the adaptation handling code to a specific form. It can either consist in library calls, or be template-generated, it can result from instrumenting the application or as a side effect of using explicit code structures/library primitives in writing the application. The approaches clearly differ in the degree of user intervention required to achieve dynamicity.

3. Examples of the Abstract Decomposition

In order to better explain the abstract adaptation model, we instantiate the model in two different applications, and discuss the meaning that actions and phases in the model assume.

3.1. Task Farming

We exemplify the abstract adaptation schema on a task-parallel computation organized around a centralized task scheduler, continuously dispatching works to be performed to the set of available processing elements. For this kind of pattern, both a performance model and a balancing policy are well known, and several different implementations are feasible (e.g. multi-threaded on SMP machines, or processes in a cluster and/or on the Grid). At steady state, maximum efficiency is achieved when the overall service time of the set of processing elements is slightly less than the service time of the dispatcher element.

Triggers are activated, for instance, when (1) the average interarrival time of task incoming is much lower/higher than the service time of the system, (2) on explicit user request to satisfy a new performance contract/level of performance, (3) when built-in monitoring reports increased load on some of the processing elements, even before service time increases too much.

Assuming we care first for computation performance and then resource utilization, the adaptation policy would be like that in Fig. 2. Applying this policy, the decide phase will eventually determine the increase/decrease of a certain magnitude in the allocated computing power, independently of the kind of computing resources.

This decision is passed to the commit phase, where we must produce a detailed plan to implement it (finding/choosing resources, devising a mapping of application processes where appropriate).

Assuming we want to increase the parallelism degree, we will often come up with a simple plan like that in Fig. 3. The given plan is the most usual one, but some steps can be skipped

- when steady state is reached, no configuration change is needed
- if the set of processing elements is slower than the dispatcher, new processing elements should be added to support the computation and reach the steady state
- if the processing elements are much faster than the dispatcher, reduce their number to increase efficiency

Figure 2. A simple farm adaptive policy

1. find a set of available processing elements $\{P_i\}$
2. install code to be executed at the chosen $\{P_i\}$ (i.e. application code, code that interacts with the task scheduler and for dynamicity handling)
3. register with the scheduler all the $\{P_i\}$ for task dispatching
4. inform the monitoring system that new processing element have joined the execution

Figure 3. Plan for increasing resources.

depending on the implementation. For example, a multithreaded program executing on a SMP architecture does not require the code to be installed (step 2). The order may also be different, e.g. swapping steps 3 and 4. Actions listed in the plan exploit mechanisms provided by the implementation, for instance to either fork new threads, or stage and run new processes or even ask for a larger processing time share (on a multiprogrammed system with QoS control at the system level). The list of steps in the plan is also customized w.r.t. application implementation. As an example, whenever computing resources are homogeneous, step 1 is quite simple, while it will require a specific effort to select the best execution plan on heterogeneous resources.

Once the detailed plan has been devised, it has to be executed and its actions have to be orchestrated, choosing proper timing in order that they do not to interfere with each other and with the ongoing computation.

Abstract timing depends on the implementation of the mechanisms, and on the precedence relationship that may be given in the plan. In the given example, steps 1 and 2 can be executed in sequence, but without internal constraint on timing. Step 3 requires a form of synchronization with the scheduler to update its data, or to suspend all the computing elements, depending on actual implementation of the scheduler/worker synchronization. For the same reason, execution of step 4 also may/may not require a restart/update of the monitoring subsystem to take into account the new resources.

3.2. Fast Fourier Transform

The Fast Fourier Transform can be implemented as a parallel SPMD code that distributes the matrix by lines. It alternates local computation and global matrix transposition steps. A performance model is known that predicts the optimal number of processors for such an application, depending on their power and the cost of communications. The code can thus be made adaptive, by spawning processes when new processors become available. Similarly, when some allocated processors are reclaimed by the operating system, concerned processes have to be safely terminated first. Thanks to the abstract model for dynamic adaptation, such behaviour can be easily designed.

The policy is composed of the following two statements: when the trigger notifies of available processors, and if the optimal number of processors is not overflowed, then the application decides to start new processes; when the trigger notifies that some used processors are reclaimed, some of the processes will be stopped. Given this decision, the commit phase produces a plan. The plans for the two kinds of adaptation are given on Fig. 4 and 5.

This example shows that the implementation mechanisms may depend on several aspects of the application. For example, redistributing a matrix is strongly dependent on the application and its implementation. On the other hand, preparation of the environment may require for example starting daemons (when using LAM-MPI communications), but it is not strictly related to the application code.

1. prepare the environment for the newly recruited processors (start daemons, stage-in files, etc.)
2. spawn processes to be executed by the new processors
3. fix connections between processes such that the new ones can communicate with the others
4. redistribute the matrix in order to balance the load amongst the whole set of processes

Figure 4. Plan for spawning processes.

1. redistribute the matrix in such a way that the terminating processes do not hold any part of the matrix anymore
2. fix connections between processes in order to exclude the processes that are terminating
3. effectively terminate the concerned processes
4. clean everything that has been previously installed specifically for the application

Figure 5. Plan for removing processes.

The mechanisms also impose various constraints on the timing phase of the abstract model, depending on their implementation. This is the case for action 2 of the plan for spawning processes (Fig. 4) that creates the processes. For an MPI application this action can be implemented either the standard way, with the `MPI_Comm_spawn`, or in an ad-hoc way if the developer requires finer control over process creation. The former approach requires synchronization of already running processes, whereas the latter may not.

4. AFPAC: A Generic Framework for Developers to Manage Adaptation

The AFPAC framework [6] focuses at present on adaptability of parallel components. Its approach consists in defining the modifications that should be applied to an existing component in order to make it able to adapt itself. Its concrete architecture (Fig.6) can be seen as a specialization of the abstract model of Fig. 1 as follows. Indeed, policy, planner and actions entities implement respectively the policy, plan and mechanisms phases of the abstract model; the timing phase of the abstract model is split over both the executor for handling the control flow and the coordinator for the synchronization with the application. AFPAC does not make appear explicitly the trigger phase as it is considered as an interface, whereas the service entity, modelling the application, has no counter-part in the abstract model. As shown in Fig.6, the decider glues the policy to the external probes in the same way that the decide phase aggregates the trigger and policy phases in the abstract model. The same kind of matching applies between the executor entity and the execute phase.

In the case of a parallel component, the service is implemented by a parallel algorithm. At runtime, it contains several execution threads distributed over a collection of processes. The AFPAC framework does not impose any constraint on communications between threads.

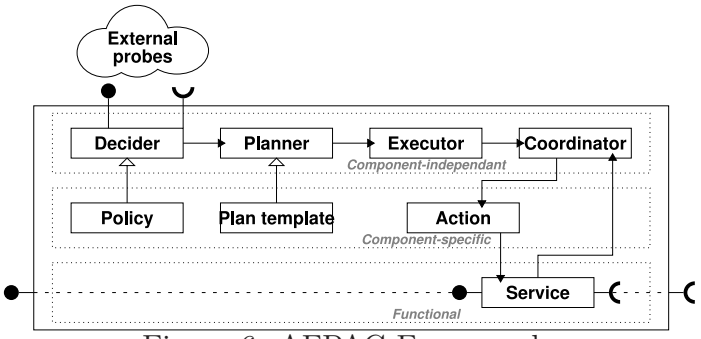


Figure 6: AFPAC Framework.

At the current state, the AFPAC framework includes two coordinators. The first one executes sequential actions and does not impose any synchronization constraint with the service. It is somewhat an empty coordinator. The other coordinator aims at executing parallel actions in the context of the execution threads of the service. To do so, it requires to suspend the execution threads at a state from which such actions are allowed to be executed. Such a state is called an adaptation point. In the case of parallel codes, adaptation points must be related in order to build a global state that satisfies some consistency model. For example, in the case of SPMD codes, such a consistency model may state that all threads should execute the action from

the same adaptation point. This problem has been further discussed in [6]; an algorithm has been proposed in [7] for implementing such a coordinator that looks for adaptation points in the future of the execution of the service. It is especially suitable for SPMD codes such as the ones using MPI (e.g. the Fast Fourier Transform example given in Sec.3.2).

The AFPAC framework gives full control over dynamic adaptation to the developer. Consequently, the developer is responsible for designing and implementing the policy, plan template and action entities. In the same way, he/she has to place manually adaptation points within the source code of the service as additional statements. Nevertheless, extra preparation of the component (such as generation of annotations required by the coordinator) is done automatically thanks to aspect-oriented programming. Thanks to this semi-automated modification and to the separation of concerns, AFPAC can be used to make adaptable existing legacy codes at a low development cost.

5. ASSIST: Managing Dynamicity Using Language and Compilation Approaches

ASSIST applications are described by means of a coordination language, which can express arbitrary graphs of (possibly) parallel modules, interconnected by typed streams of data. A parallel module (*parmod*) coordinates a set of concurrent activities, which are performed by *Virtual Processes* (VPs). VPs execute a set of sequential activities on their input data and internal state, activities that are selected on item arrival from the input streams. The sequential functions can be programmed using standard sequential languages (C, C++, Fortran).

Overall, a *parmod* may behave in a data-parallel (e.g. SPMD/for-all/apply-to-all) or task-parallel way (e.g. farm, pipeline), and it can nondeterministically accept from one or more input streams a number of input items, which may be decomposed in parts and used as function parameters to activate VPs. A parmod may also exploit a distributed shared state, which survives between VP activations related to different stream items. More details on the ASSIST environment can be found in [11,2].

An ASSIST module (or a graph of modules) can be declared as a component, which is characterized by provide and use ports (both one-way and RPC-like), and by *Non-Functional* ports. Among the non-functional interfaces there are those related to QoS control.

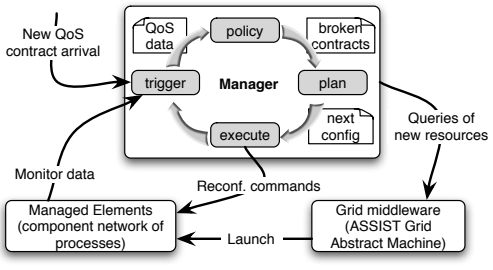


Figure 7: ASSIST framework.

At any moment during an ASSIST application run, components can be assigned a new QoS contract, e.g. specifying a performance requirement. In order to fulfil the contracts, the component framework continuously adapts component configurations, in terms of parallelism degree, and process mapping [3]. The adaptation mechanism relies on automatic user code instrumentation, and on a hierarchy of Application Managers [1].

Each component has a Component Adaptation Manager (CAM) entity coordinating its adaptation. An Application Manager (AM), possibly distributed, enforces QoS of the application in the whole, by coordinating and leveraging on CAMs. As sketched in Fig. 7, ASSIST implements the abstract adaptation schema by organizing its leaves, left to right (compare with Fig. 1) in an autonomic control loop. CAM managed entities are processes within a component, while the AM applies the abstract model to application components. In the following we describe the CAM case.

The trigger functionality has to (1) collect a stream of monitoring data from the running program, as a feedback to the autonomic behaviour of AMs, and (2) to react to QoS contract changes when they trigger the need for adaptation.

The policy phase in Fig. 7 evaluates a component performance model over the monitoring data, to find out the amount/allocation of resources that can match the assigned QoS contract. In the case the QoS contract is broken, the decide phase will set a target for the commit phase, e.g. the additional amount of required computing power. The ASSIST compiler synthesizes the performance model from static information on the parallel pattern exploited by the component. Application programmers can also override standard performance models with custom ones.

The plan phase in Fig. 7 re-conveys component performance within its contractually specified values by exploiting the set of actions available as mechanisms. Plan templates are instantiated as partially ordered sets of actions, which are performed according to the schedule provided by timing. ASSIST implements two layers of adaptation mechanisms: parallelism degree management (add or remove resource to/from computation), and computation (VP) remapping, with associated data migration and global state consolidation.

The timing functionality, not shown in Fig. 7, involves a distributed agreement among a set of VPs on the point where the reconfiguration must happen. In ASSIST the migration process can be performed in so-called *reconf-safe* points [3], i.e. points in the application code where the distributed computation and state are known to be consistent, and can be efficiently synchronized. Placement and use of reconf-safe points are automated, so that different mechanisms available to the execute phase (reconfiguration commands in Fig.7) automatically get the appropriate kind of synchronization.

The execute functionality thus exploits support code built within the VPs, and coordinates it with services provided by the component framework to interface to Grid middleware (e.g. for resource recruiting).

Observe that all the code needed to perform the timing and execute phases is automatically generated by the ASSIST compiler, which instruments the application code in a fully transparent manner for the application developer. ASSIST reconf-safe points are designed to exploit synchronization points already needed to ensure the correctness of the parallel application code. Moreover, the ASSIST high-level structured nature enables the compiler to automatically select the optimal implementation of mechanisms for each application and reconf-safe point. For instance, no state migration code is inserted for stateless computations, and depending on the parallelism pattern (e.g. stream versus data parallel), VPs involved in the synchronisation can be a subset of those within the component being reconfigured.

In this way ASSIST adaptive components run with no overhead with respect to non-adaptive versions of the same code, when no configuration change is performed [3].

6. Conclusions

We have described a general model to provide adaptive behaviour in Grid-oriented component-based applications. The general schema we have shown is independent of implementation choices, such as the responsibility for inserting the adaptatation code (either left to the programmer, as it happens in the AFPAC framework, or performed by exploiting knowledge of the high level program structure, as it happens in the ASSIST context). The model also encompasses user-driven as well as autonomic adaptation.

The abstract model helps in separating application and run-time programming concerns of adaptation, exposing adaptive behaviour as an aspect of application programming, formalizing

the concerns to be addressed, and encouraging an abstract view of the run-time mechanisms for dynamic reconfiguration.

This formalization gives the basis for defining a methodology. The given case studies provide with valuable clues about how to solve different concerns, and how to identify common parts of the adaptation that can be generalized in support frameworks. The model can be thus also usefully applied within other programming frameworks, like GrADS, which do not enforce a strong separation of adaptivity issues into design and implementation.

We expect that such a methodology will lead to more portable and understandable adaptive applications and components, and it will promote layered software architectures for adaptation, simplifying implementation of both the programming framework and the applications.

Acknowledgments. This research work is carried out under the FP6 Network of Excellence *CoreGRID* funded by the European Commission (Contract IST-2002-004265), and it was partially supported by the Italian MIUR FIRB project *Grid.it* (n. RBNE01KNFP) on High-performance Grid platforms and tools.

References

[1] M. Aldinucci, S. Campa, M. Coppola, M. Danelutto, D. Laforenza, D. Puppin, L. Scarponi, M. Vanneschi, and C. Zoccolo. Components for high performance Grid programming in Grid.it. In V. Getov and T. Kielmann, editors, *Proc. of the Workshop on Component Models and Systems for Grid Applications (June 2004, Saint Malo France)*. Springer, January 2005.

[2] M. Aldinucci, M. Coppola, M. Danelutto, M. Vanneschi, and C. Zoccolo. ASSIST as a research framework for high-performance Grid programming environments. In J. C. Cunha and O. F. Rana, editors, *Grid Computing: Software environments and Tools*. Springer, 2005. (to appear, draft available as TR-04-09, Dept. of Computer Science, University of Pisa, Italy, Feb. 2004).

[3] M. Aldinucci, A. Petrocelli, E. Pistoletti, M. Torquati, M. Vanneschi, L. Veraldi, and C. Zoccolo. Dynamic reconfiguration of grid-aware applications in ASSIST. In *11th Intl Euro-Par 2005: Parallel and Distributed Computing*, LNCS, Lisboa, Portugal, August 2005. Springer. To appear.

[4] Françoise Baude, Denis Caromel, and Matthieu Morel. On hierarchical, parallel and distributed components for Grid programming. In V. Getov and T. Kielmann, editors, *Workshop on component Models and Systems for Grid Applications*, ICS '04, Saint-Malo, France, June 2004.

[5] F. D. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application-level scheduling on distributed heterogeneous networks. In *Supercomputing '96: Proc. of the 1996 ACM/IEEE Conf. on Supercomputing (CDROM)*, page 39, 1996.

[6] J. Buisson, F. André, and J.-L. Pazat. Dynamic adaptation for grid computing. In *European Grid Conference 2005*, Amsterdam, February 2005.

[7] J. Buisson, F. André, and J.-L. Pazat. Enforcing consistency during the adaptation of a parallel component. In *The 4th Int.l Symposium on Parallel and Distributed Computing*, July 2005.

[8] B. Ensink, J. Stanley, and V. Adve. Program control language: a programming language for adaptive distributed applications. *Journal of Parallel and Distributed Computing*, 63(11):1082–1104, November 2003.

[9] M. McIlhagga, A. Light, and I. Wakeman. Towards a design methodology for adaptive applications. In *Mobile Computing and Networking*, pages 133–144, May 1998.

[10] S. Vadhiyar and J. Dongarra. Self adaptability in grid computing. *International Journal Computation and Currency: Practice and Experience*, 2005. To appear.

[11] M. Vanneschi. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing*, 28(12):1709–1732, December 2002.

Load Balancing Support for Grid-enabled Applications

S. Rips[a]

[a]Heinz Nixdorf Institute, University of Paderborn, 33102 Paderborn, Germany

Executing parallel applications in Grid environments, often leads to poor efficiency values due to different compute power and different networks. Hence, new strategies are required for reducing runtime. Especially, load balancing must be adapted to the characteristics of this heterogeneous environment. We developed a supporting module, which explores the used environment and supplies the load balancer with information. The load balancer considers this additional information while partitioning the data and operations. This leads to execution patterns with minimised communication and reduced idle times caused by blocking sends/receives.

1. Introduction

Grid Computing enables a seamless access to a large number of compute, storage, network, and other resources, which are used by different communities from industry and academia for solving complex problems. Aggregation of multi-site resources allows the processing of large-scale problems that are too complex for traditional clusters and other parallel machines [6]. However, despite the tremendous research efforts in developing Grid middleware and architecture components, a lack of running Grid-enabled applications is still noticeable. Difficult installation of Grid middleware and resource discovery is one of the major problems. Moreover, users are forced to modify and to adapt their applications to the heterogeneous Grid world, which significantly differs from the traditional cluster environment built around homogeneous processing elements and networks. A direct execution of the existing parallel applications in Grid environments often leads to poor efficiency values, as the differences between the involved processing elements and the long communication times due to the slow networks between Grid sites are not sufficiently considered. In order to solve this problem, decisive system software components such as the load balancer have to be aware of the characteristics of the current Grid environment. Therefore, a supporting module for any kind of load balancers has been developed, which explores the current environment and supplies the load balancer with this knowledge. The load balancer considers this additional information while partitioning the data and operations and thus leads to execution patterns with minimised intra-partition communication and reduced idle times due to blocking sends/receives. After a certain runtime the overhead created by the environment analysis is neglected as compared to the speed-up and efficiency values achieved with the optimised partitioning and distribution.

This paper presents the core method for monitoring and discovering the network topology and the characteristics of the involved compute sites. Subsequently, the developed architecture and the current implementation of the prototype are described. Finally, a set of performance measurements shows the quality of the developed solution by considering CFD (Computational Fluid Dynamics) applications as an example.

2. Related Work

The structure of the Grid comprises characteristics of homogeneous as well as heterogeneous systems, loosely coupled as well as tightly coupled systems.

Load balancing strategies aim to adapt the load optimally to the environment. However, they mainly consider the application running on a parallel, homogeneous system. Only a few methods address also the special characteristics of the underlying system.

Zaki et al. [10] consider different processor speeds and distribute the load adequately. However, processor speeds are obtained by a profiling run and they assume full connectivity among the processors, with uniform latency and bandwidth.

Hendrickson and Devine [4] review the major classes of dynamic load balancing (DLB) approaches. They point out that for heterogeneous systems, different amounts of computing power and memory should be considered. Additionally, they emphasise that network connections with different speeds are important for a DLB strategy. However, a solution is not proposed.

Kielmann et al. [5] emphasise that a collection of clusters can be seen as a hierarchical system. They use a tree topology to do load balancing for divide-and-conquer applications. However, they do not take different PE characteristics into account.

Willebeek and Reeves present in [9] a hierarchical balancing method (HBM). HBM is an asynchronous, global approach which organises the system into a hierarchy of subsystems. The strategy has been implemented on a hypercube system. Due to its hierarchical structure, it is not necessary to perform any analysis of the topology at the beginning as well as modifications during runtime.

Especially [5] and [9] show the importance of considering the underlying network.

We did not found any methods in the literature that detect the hierarchical structure of a Grid environment and use the gained results to optimise middleware, as e.g. load balancing, specifically for this structure.

The next section describes the analysis of the system in order to build a base for load balancing decisions. The first step is the analysis of the underlying network that results in a hierarchical subsystem, where each subsystem indicates small communication times inside and slower ones outside. The following discovery of each node's capacities and therewith the capacities of each subsystem is a further step to support load adaptation to a heterogeneous system.

The resulting structure constitutes an appropriate basis for the detection of load imbalance and for minimising the amount of PEs that participate on load balancing.

3. Grid Environment Discovery

3.1. Network

Based on monitoring the network, the system is organised into a hierarchy of subsystems that reflects the current system status independent of physical connections. Each level of the hierarchy represents a magnitude of communication speed.

Our method to create this hierarchical representation follows a distributed approach without having a global decision instance. This enables scalability, which is particularly important when using thousands of processing elements (PEs) as in the Grid.

Communication times, resulting from measurements, are used to set up the hierarchy levels. For the lowest level, each PE_k builds a basic subsystem Sub_k out of all PEs $PE_j, 1 \leq j \leq n$ with communication times $t(k,j)$, with:
Let $T^k_{com} = (t(k,i_1),...,t(k,i_{n-1}))$, where $t(k,i_j) \leq t(k,i_{j+1})$ and $i_j \neq k$.
Then, a subsystem Sub_k is defined as:
$Sub_k = \{PE_{i_1},...,PE_{i_l}, PE_{i_p} \in T^k_{com} | l = min\{1,...,n-1\}, s.t. \frac{t(k,i_l+1)}{t(k,i_l)} \geq threshold > 1\}$,
with $threshold$ set to any fixed value.

This means, a subsystem is built out of PEs with smallest communication times. The ratio of the

PE with the smallest communication time outside the subsystem and those with the highest time inside the subsystem, is greater than $threshold$. The threshold indicates a jump in the list of sorted communication times, i.e. a next level of communication times.

By this procedure, magnitudes of communication speeds are considered. By setting the threshold to a small value ($1 < threshold < 1.8$), the resulting granularity of the hierarchy is much finer than setting this parameter to a high one.

Specific PEs are designated to control the hierarchy discovery of the next level. These PEs (master nodes) again combine subsystems to new ones based on their interconnection speeds as described for the lowest level. This proceeds until the whole system has been analysed.

During runtime, synchronisation points of the application are used to monitor the current network situation. If considerable changes are detected our system responses by adapting the subsystem hierarchy.

3.2. Compute Nodes

Besides this fundamental analysis of the network, support for Grid-enabled applications implies also the inspection of the PEs' capacities.

We fulfil this requirement by monitoring the processing speed of each PE while the application is running. This is done by measuring the time that is spent on calculating an application's load unit (e.g. a cell calculation in CFD applications). The gathered information is used to determine the optimal percentage of load each PE should work on. Based on this information, the imbalance for each subsystem is calculated.

4. Rebalancing

Rebalancing load between a few adjacent PEs can lead to a sufficient common load balance. Not all PEs must be involved in the load balancing procedure. Our module provides an appropriate subsystem of the hierarchy to the load balancing component of the application. All PEs belonging to a subsystem that is balanced in total but whose sub-subsystems are unbalanced, have to do load balancing.

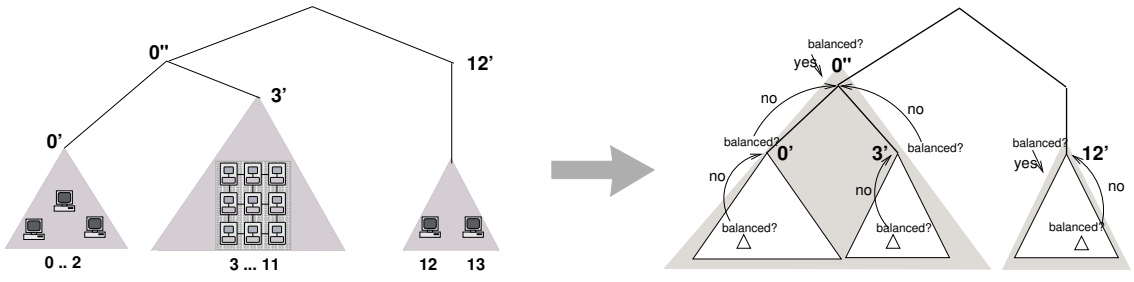


Figure 1. Subsystem hierarchy and detection of load imbalance

Figure 1 shows an example of detection of load balancing participants. Some PEs of the lowest level are over- or underloaded and send a request to their master nodes. The master nodes (here: 0', 3', 12') check whether the subsystem is balanced or not, i.e. whether the current load of the

subsystem differs from its optimal load. If it is balanced (here: subsystem 12') its associated nodes (here: PEs 12, 13) must do load balancing within the subsystem.

If the subsystem is not balanced, the request is passed to the master node of the next higher level (here: 0' and 3' pass requests to 0''). This proceeds until a balanced subsystem is found. Since the whole system is always balanced, this process stops at the latest at the highest level.

In the example, subsystem 0'' with PEs 0, ... 11 has to rebalance its load between its associated nodes but independently from subsystem 12'.

5. Architecture

Two goals determined the design of the architecture: transparency and minimising necessary changes of the application code. The last point led to the usage of MPI [2]. This decision was forced by the fact, that a lot of parallel codes of high performance applications as well as parallel load balancing tools use MPI to implement communication between processes. Furthermore, MPI provides a profiling interface. It enables the execution of extra code when calling designated MPI functions. This procedure is fully transparent for the MPI application.

The supporting module consists of two components (see figure 2). The communication monitor (CM) observes the network whereas the application monitor (AM) considers the capacity of the PEs.

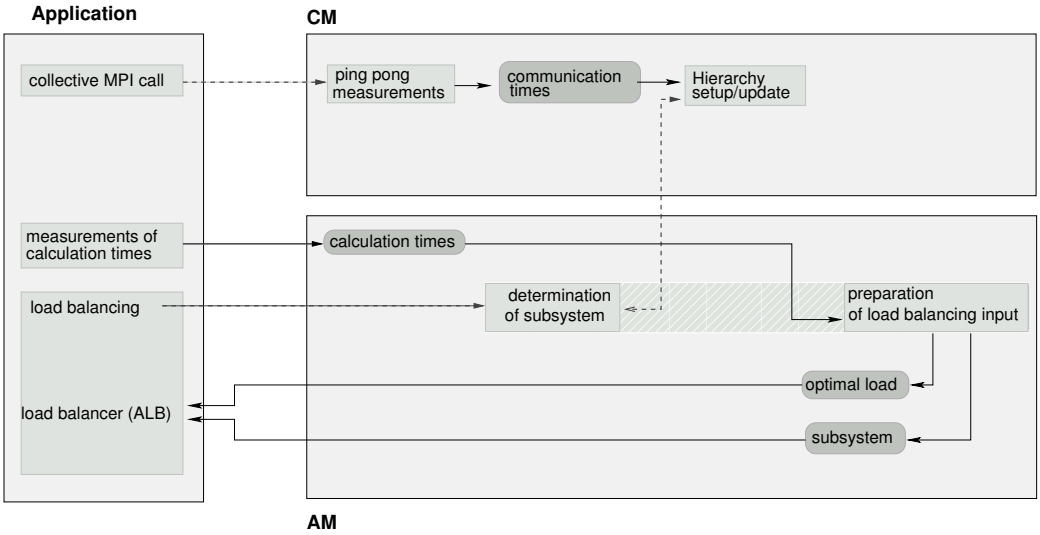


Figure 2. Architecture of Monitoring System

The CM uses the MPI profiling interface to make network monitoring transparent for the application. Thereby, no changes of the application code have to be done. It is sufficient to re-link the application to an extra library. The CM automatically sets up the subsystem hierarchy at program start based on measurements of ping-pong tests. These tests are done under the control of a pre-calculated communication schedule. It avoids conflicts in the send–receive procedure and minimises the number of ping-pong rounds.

The size of the messages is set to 1 kB. Tests with different message sizes (1 kB, 64 kB and 64 MB) have shown that the number of bytes sent is nonrelevant when detecting magnitudes of

communication times.

Whenever a collective MPI routine is called by the application, the CM performs communication measurements as done at the beginning. These measurements are used for subsystem updates, which are necessary before calling load balancing.
The application has access to the hierarchy via defined interfaces. The CM is realized as a library that provides a small API to the developer.

The integration of the AM requires only few modifications of the application code. Runtime measurements of a calculation unit must be done. The results must be passed together with load information via an API to the AM. These values are used by the AM to calculate the optimal load rate.

When load balancing is called by the application, the AM initiates the determination of the subsystem in which the rebalancing has to be done. First, it assigns the CM to check the hierarchy's up-to-dateness. If significant network changes occurred, the CM updates it. Based on this hierarchy, the AM then determines the PEs that participate on the same load balancing process as itself.

The preparation of load balancer input is the final task of the AM. Based on the calculation times provided by the application, together with the current load, the optimal load is calculated and passed to the application. This information is used as parameters for the application's load balancer (ALB).

The monitors on the different PEs have their own view on the system. Each CM/AM knows all members of its own subsystem and, in case of being a master node, keeps load information of its subsystem and the lower level subsystem. This information is sufficient when inspecting the balance from leaves to root. The absence of a global instance ensures scalability.

6. Prototype

For our prototype we use mesh based applications, e.g. CFD codes, that use domain decomposition for load balancing. In this area, several dynamic load balancing tools have been developed, but only few of them (e.g. Jostle [8], ParMetis [7]) can handle information about the required partition sizes, as provided by our software.

When load balancing is initiated, the subsystem hierarchy provided by the CM is extended by the load information delivered by the AM. An optimal load amount is determined and passed to the application's load balancer that calculates a new partitioning.

The determination of the participating PEs in load balancing is done by our module. It detects an appropriate subsystem and passes the corresponding PEs to the load balancer.

For Jostle, this is done by creating a new MPI communicator containing the current load balancing partners. Jostle performs the calculations of new partitions, with sizes given by the AM, on the PEs of this new communicator.

7. Performance evaluation

Runtime measurements were done on a PC-Cluster with InfiniBand [1] network and two processors (64-bit Intel Xeon) on each node, communicating via shared memory. Our algorithm is able to detect this two level hierarchy (shared memory and InfiniBand communication).

Tests with PACX-MPI[3] had shown that further hierarchy levels are detected by our algorithm. With support of PACX-MPI, MPI-conforming parallel applications can run on a Computational Grid. It enables the coupling of several MPI applications running as a single virtual machine.

A simple simulation program representing the behaviour of an adaptive CFD code was used to perform the measurements. The program calculates initially 1,000 mesh cells and ends up with 100,000 mesh cells. In the test version without our support module the number of mesh cells passed to Jostle is the same for each PE.

We used 2 x 20 PEs, i.e. 20 nodes, each with two PEs. Heterogeneous compute power has been simulated by assigning extra work to some PEs. For the homogeneous version, this extra work stayed away.

	heterogeneous		homogeneous	
	with LB support	no LB support	with LB support	no LB support
total runtime	88.25 sec	260.17 sec	89.37 sec	93.28 sec
idle times[1]	< 3 sec	< 29 sec	< 2.3 sec	< 2 sec
ALB Jostle	3.1 sec	8.5 sec	1.7 sec	8.2 sec
LB overhead	7.5 sec		9.48 sec	

[1] due to blocking send/receives

The results shown in this table expose a high potential of our method. The results of the homogeneous version illustrate the runtime benefit achieved by utilising the hierarchy. Load balancing is mostly done between PEs on the same node (using shared memory communication). This leads to much lower runtimes of jostle compared to the version without support where all PEs are involved in each load balancing step.

The heterogeneous version shows a big gain obtained by our software. The adaptation of load to each PE's capacity shortens the runtime to one-third. The idle waiting times, that decrease performance, can be reduced to nearly the tenth part.

Although the test runs had been done in an environment with very small sized subsystems at lowest hierarchy level (the two PEs, residing on one node), better results are gained compared to runs without our support. More improvements can be expected in Grids with more PEs in low level subsystems. where load can be redistributed between more than two PEs.

Reducing the overhead for load balancing support, e.g. by optimising the code, will lead to further runtime reductions.

8. Conclusion

A tool has been presented that supports load balancing for Grid-enabled applications by analysing the environment. Load balancing decisions are based on a hierarchical structure of subsystems that represents different classes of communication speeds. By doing load balancing on a subset of PEs with fast connections, the load balancing overhead is significantly reduced. We showed that adapting the load to the different capacities of the compute nodes, leads to further drastically runtime improvements.

The integration of the presented tool requires only few modifications to the application code. Since we use no global instance, our tool is able to support applications using thousands of PEs.

References

[1] Infiniband. http://www.infinibandta.org/home.

[2] MPICH homepage. http://www-unix.mcs.anl.gov/mpi/mpich.

[3] PACX-MPI homepage. http://www.hlrs.de/organization/pds/projects/pacx-mpi.

[4] Bruce Hendrickson and Karen Devine. Dynamic load balancing in computational mechanics. In *Computational Methods in Applied Mechanical Engeneering*, volume 184, pages 485 – 500, http://www.cs.sandia.gov/ kddevin/main_publist.html, 2000.

[5] Thilo Kielmann, Henri E. Bal, Jason Maassen, Rob van Nieuwpoort, Lionel Eyraud, Rutger Hofman, and Kees Verstoep. Programming environments for high-performance Grid computing: the Albatross project. *Future Generation Computer Systems*, 18:1113–1125, 2002.

[6] Paul Messina. Distributed supercomputing applications. In Ian Foster and Carl Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2003.

[7] Kirk Schloegel, George Karypis, and Vipin Kumar. A unified algorithm for load-balancing adaptive scientific simulations. In *Supercomputing 2000*, 2000.

[8] C. Walshaw and M. Cross. Multilevel mesh partitioning for heterogeneous communication networks. *Future Generation Comput. Syst.*, 17(5):601 – 623, 2001.

[9] Marc H. Willebeek-LeMair and Anthony P. Reeves. Strategies for dynamic load balancing on highly parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(9):979 – 993, 1993.

[10] Mohammed Javeed Zaki, Wei Li, and Srinivasan Parthasarathy. Customized dynamic load balancing for a network of workstations. *Journal of Parallel and Distributed Computing*, pages 156 –162, 1995.

NewsGrid: Infrastructure and Services

S. Geisler[a], G.R. Joubert[a]

[a]Department of Computer Science, Clausthal University of Technology, Julius-Albert-Straße 4, 38678 Clausthal-Zellerfeld, Germany, E-mail: {geisler, joubert}@informatik.tu-clausthal.de

1. Introduction

TV stations need access to news video material in order to compile news casts. Such material is collected twenty-four hours per day seven days per week from different sources such as news agencies and own correspondents in analogue and/or digital form via various networks (telephone, satellite transmissions, mail, etc.). It cannot be predicted in advance which of this material will actually be used. Thus all material received by a TV station must be catalogued and stored in archives. On average only a small percentage of the archived material is actually used. The retrieval of material from such libraries is completely dependent on the quality of the cataloguing procedures.

To-day the quality of digital videos equals or even surpasses that of broadcast quality analogue material. Digital videos have the added advantage that they can be stored in media databases and be transferred without loss of quality to transmitting stations. Such digital archives can be made accessible to many TV stations, thus spreading the maintenance cost. Digital repositories furthermore open up the possibility to use content based video and image retrieval techniques. Thus individual stations need retrieve only the material they actually need for particular news casts. The net result of these advantages is that digital repositories are increasingly replacing analogue ones.

In [1] a world wide NewsGrid was proposed. The basis of the grid is a distributed digital archive, supplying versatile services for storing, searching and retrieving, downloading, and post-processing of news videos. It completely changes the way in which news video material is distributed, as well as the workflow for news editors compiling news casts. The video content retrieval process can, in addition to manual indexing techniques, employ dynamic feature extraction techniques. This reduces the cost of indexing videos manually. A disadvantage is that it requires high-speed computing resources.

Within the NewsGrid framework news videos are not distributed to subscribing TV stations. Instead news suppliers make available information about the content of available material. The users, i.e. TV stations, retrieve the desired material on a subscription or pay per item basis directly from the news suppliers' video repositories from all over the world. This results in huge savings in bandwidth and storage capacity compared to current procedures as only the material actually needed needs to be distributed. Users will, however, need fast and effective search mechanisms to retrieve specific news material.

The major potential advantages for news agencies are a huge reduction in information broadcast costs and improved statistics on which news items are actually used by clients. The use of the proposed NewsGrid requires fundamentally new marketing and distribution strategies by news suppliers. Instead of distributing news content only, information about available news material will have to be compiled and distributed or made available on the Internet.

Essential requirements for the implementation of the proposed NewsGrid are:

- Reliable and easy to use grids comprising powerful database servers
- User friendly, effective, efficient and interactive retrieval mechanisms

- Compact information being made available on news – or for that matter other – content contained in supplier databases.

2. Compilation of Newscasts

TV news casts are compiled by combining new video material received from news agencies, own correspondents or correspondents of other TV stations, with historic or reference material retrieved from archives. The classic approach is to record news videos in analogue broadcast quality on magnetic tapes. Each tape is manually catalogued with key words and content descriptions and stored in a local library. The retrieval of video material is achieved by using the catalogue and a final visual screening of selected items. This is a highly inefficient, labour and cost intensive and often ineffective process as all these actions require human actions. This holds for both the cataloguing and retrieval processes.

In the case of digital video material the repository may be realised with a DBMS (Database Management System) adapted to video content storage and retrieval, e.g. OVID [2]. The retrieval is achieved by use of catalogued descriptions and/or key words. The screening of video material for final selection can be considerably simplified [3].

Digital video databases furthermore allow the application of multimedia technologies utilising content based retrieval (CBR) methods. CBR, also referred to as feature extraction methods, have the potential of making the time and cost intensive task of manually indexing each video clip superfluous.

3. NewsGrid

The NewsGrid concept (Fig. 1) described in [1] is based on the use of dynamic feature extraction. This makes it possible to employ powerful and flexible retrieval techniques to search for particular video content. The main advantages offered by NewsGrid are that:

- News suppliers need not distribute their material to users as they can directly access and search suppliers' video repositories
- Users need only pay for material they actually use
- News agencies can gain better insight in what material is actually used by which user(s) and can thus improve their news collection activities.

Disadvantages of this approach are:

- The fact that CBR methods are still objects of research; presently available methods are promising, but still limited in their effectiveness
- The workflow of news editors is changed and they must learn to use grid technologies
- The marketing strategies of news suppliers must be adapted.

For the NewsGrid model to become accepted in the market place it is thus essential that:

- Effective CBR methods are available
- The high computational demands of CBR methods are met
- Suitable grid services to support users are available.

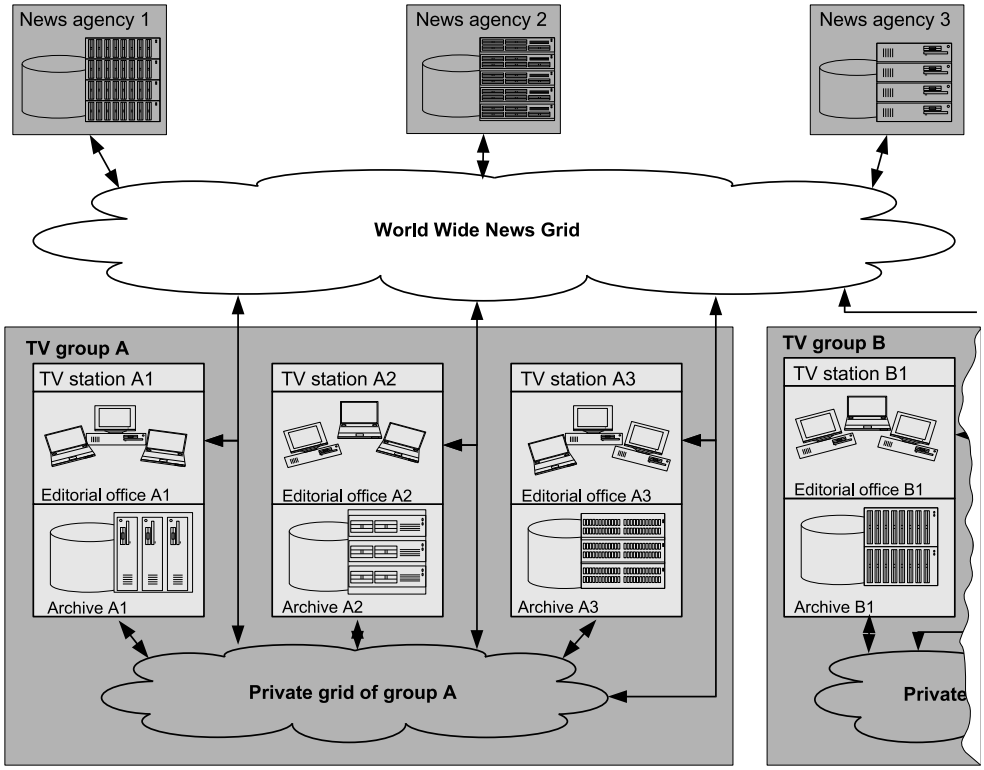


Figure 1. Structure of the NewsGrid. Several private grids (intragrids) of TV station groups are connected to the world wide NewsGrid.

To cope with the high computational demands, efficient and effective content based search algorithms are required. First results were discussed in [1]. The processing infrastructure must make available powerful parallel processing resources to achieve acceptable response times, as well as grid services that support users in their quest for information. In [1] an outline of infrastructure requirements as well as parallel processing models was discussed. In this paper the NewsGrid infrastructure and the required grid services will be explored in more detail.

4. Infrastructure

In [4] different techniques were compared in order to formulate fast parallel search methods and suitable parallel architectures. The results were obtained through simulation of the performance of various prototype configurations. The best performance is gained with a cluster of dual processor nodes and coarse-grained parallelisation. An important aspect is the communication networks connecting nodes of clusters in such a way that efficient video retrieval can be attained by a large number of users.

Within a NewsGrid news videos are stored in a relatively small number of distributed archives, thus substantially reducing redundancy. This greatly reduces the overall cost of distributing, indexing and storing archive material. The indexing process can thus be better formalised resulting in more consistent results.

The infrastructure of the whole grid is organised hierarchically (fig. 1). Each TV station has its own archive, and each TV group can have its private grid. This saves communication bandwidth. Within a smaller group higher network capacities can be implemented while the world wide grid

usually offers a lower transfer speed. Additionally downloading the same video several times should be avoided to save time and reduce costs.

Every time a participant of the grid obtains a video from another TV group or a news agency the downloaded file is temporarily stored in a local archive. If the video is accessed a second time from the same station or group, the grid software detects the video file in the local cache. Before starting a download, it is checked, whether the original material has been updated or changed. If not, the download is started from the local media server instead of from remote. The file is deleted from the cache, if it has not been accessed for a longer period. The relevance of a video evaluated by humans or automatically determined number of accesses may be used to implement a replacement policy.

A requirement for the successful implementation and use of a NewsGrid is that a set of standard methods and protocols must be agreed and adhered to. Present grid standards as well as MPEG-7 and MPEG-21 form a good starting point for defining such standards. In [5] a scheme was proposed that allows the management of news feeds stored in a cluster based database. This scheme can also be used as a basis for defining schemes that can be used in NewsGrid.

It is assumed that a news header, containing static and dynamic information, is supplied. MPEG-7 can be used to achieve this. It is also possible to supply such descriptions as separate information added to any stored videos. In general the static information supplied includes, for example Date, Time, Title, Keywords/Summary, Source, Copyright/Price, Characteristics for content based retrieval, e.g. colour moments of key frames, file name or URL of video file, etc. Additionally dynamic information can be added, such as sample sketches and images, definition of regions of interest (ROI), etc. Such descriptions can be used to define a search vector or Object Description Vector (ODV) that is defined for each video clip stored in the database. The search vector can be enhanced by adding key words and textual descriptions. These must be manually added when the video clip is stored in the database.

5. Video Storage and Retrieval

With the increasing number of available news videos the available methods using static data to define object description vectors (ODVs) are becoming increasingly inadequate for information retrieval tasks. The addition of textual descriptions and/or key words is becoming too costly. The storage and efficient and effective retrieval of video files were and are thus investigated in a number of research projects, for example VideoQ [6] and CueVideo [7]. The goal with all of these is to implement a content based search on the video data in order to avoid the expensive and inaccurate compilation of textual descriptions.

In the ODV metadata, for example title, author or date, and extracted features, such as colour, textures, shapes or motion can be stored. The presently available systems have in common that the ODVs are calculated when the video is stored in the database. As the provider cannot know what the interests of the later users will be, it is likely that significant information for the users will not be extracted and stored.

An alternate approach is to store video clips with a limited ODV containing only static data (SODV). The dynamic components of the ODV (DODV) are then calculated when the user submits his query to the video database. Thus the ODV, comprising the SODV and DODV, can only be calculated when the query of the user is known. This is an extension of a proposal by Kao for image databases [8].

Objects or persons can be found in video clips by a template matching algorithm that is performed on each image in each video clip in the database. Such systems need a huge amount of computational power. This can only be achieved with parallel systems. In the case of the proposed NewsGrid this

implies that the individual nodes accessing the video databases must have sufficient compute power available. Due to the complex and compute intensive retrieval methods this also requires that a significant parallel compute power be available across the grid.

6. The retrieval process

Instead of key words and logical operators a query can consist of a sketch or a sample image supplied by the user when he defines his search. Additionally the user can mark a region of interest (ROI), which is in most cases a figure or an object in an image. The objective is not to find exact matches, but rather to locate a set of video clips or shots that contain a frame or frames corresponding as closely as possible to the sample sketch, image or the specified region. The comparison between the query image and a video frame results in the same process as in the case of searches in image databases. In the case of an object search, template matching is used. This means the query template is compared to every equally sized region of the video frame with the same size.

7. Content based video coding

Within the NewsGrid content based video coding is used in specific situations to reduce the bitrate. This means that videos are not coded on a frame by frame basis. Instead, different objects in the video scenes are coded separately and labeled with entity identifiers.

Content based video coding can be applied to TV news casts. In the case of most TV news casts scenes are composed of a background, which never changes, an information board, which changes only at the beginning of a new theme and a speaker. The speaker is an animated object. Using this information about the scene elements a segmentation of a news cast video is possible.

In order to reduce the bitrate the background is encoded only once at the beginning of the video file, while each information board is encoded once at the beginning of each theme. Only the speaker area, which usually fills about half of the screen must be encoded with the full frame rate. Using this technique the number of pixels to be encoded can nearly be halved without any loss of quality.

A further reduction can be achieved e.g. for transmission to mobile devices. As the user is normally not interested in the background the resolution of this can be reduced by a factor of 4. Small details of the speaker are also less important, therefore the resolution can be lowered by 2 in this area. With this method the quality is only lowered in areas with low visual information and the bitrate can be reduced by over 70%. An additional aspect of mobile devices are the small displays. Content-based video coding can be used to improve readability of text by introducing user interaction. If the user selects the information board in can be displayed full-screen and the font size is increased. The content based video coding for mobile devices is described in detatil in [9].

The content based video coding can also be used for annotation and link generation. As a headline is part of the information board a text extraction can be applied to this video object. The extracted text can be used as initial key word set of the news caster scene. Similarity of information boards of different news casts can be used to automatically find candidates of related videos.

For all these use-cases additional computational power is needed. Therefore the transcoding and post-processing has to be performed on grid-ressources with high computational power.

8. Grid Services

Access to the video files made available on the repositories of NewsGrid is realised by a number of grid services [1]. In addition to the standard NewsGrid services each user must be able to start

his own programs for processing data available in repositories to which he has access. By defining different user groups and domains particular services can be made available to a user group(s).

A number of sample services needed in the case of NewsGrid are outlined here. The functionality of each service is merely exemplary and can be adapted to meet the needs of particular users:

1. File-Transfer (Download): This service enables the transcoding and downloading of a video file in combination with the specification of start and end positions.
Input parameters: Video-Id, start- and end-position, video-codec (e.g. MPEG-1/2/4, H.264), audio-codec (e.g. MP3, AAC), quality/bitrate, transmission standard (NTSC/PAL/HDTV).
Function: Recode video segment, copy recoded video to ftp-server, transmit from ftp-server to client
Result: Video file.

2. File-Transfer (Upload): With the aid of this service a new video can be uploaded and stored in the database. It is devided in the following three sub-services.

2.1. Prepare Upload: A service to prepare the upload of the new video file.
Input parameters: Author, date, time, location, title, key words, copyright.
Function: Generate database entry for new video, generate file name for new video.
Result: File name for new video.

2.2. Upload new video file: Transmit the new video file to the grid archive.
Input parameters: Video file, file name generated from sub-service prepare upload.
Function: Transfer video file via ftp from client to server, extract and classify video objects, transcode video, copy file to grid directory.
Result: Video-Id.

2.3. Generate description vector: Generates the SODV.
Input parameters: Video-Id, priority.
Function: Calculate automatically retrievable features (e.g. OCR from information board), send message to archive administrators to generate content description (using priority).
Result: Success/error.

3. Video streaming (video-on-demand service): A video is transferred to a user not with the purpose of downloading the file, but rather to view and inspect the contents.
Input parameters: Video-Id, start- and end-positions, video-codec (e.g. MPEG-1/2/4, H.264), audio-codec (e.g. MP3, AAC), quality/bitrate, transmission standard (NTSC/PAL/HDTV).
Function: A copy of the video is moved to the streaming server that manages the communication with the client. Before transmission the video is transcoded into the desired format. The user must be able to start, pause, and spool forwards and backwards, etc. as is the case with a normal video player. The appropriate parameters are made available by the video-on-demand service.
Thus only those sections of the video file to be viewed need be transferred. As the purpose is to give the user a quick insight into the video contents, the quality level can be comparatively low, which can be achieved through a stronger compression. This also reduces the amount of data to be transferred.
Result: Video stream.

4. Addition of a Description: This service is used by the archive administrators to manually add additional metadata and descriptions as well as references to related material in the database after a message from service 2.3 was received or when an update must be executed.
Input parameters: Video-Id, data for the description.
Function: Store the changed SODV in the database.
Result: Success/error.

5. Data retrieval using static features (SODV): This service enables the classic data retrieval tasks using one of the static features from the ODV. This can be directly supported by the user e.g.

in the case of key words or calculated on the client or on the server side, if the query by example technique is used. By combining different features and searching on former result sets the search can be limited to subsets of the database.
Input parameters: Feature-Id, corresponding feature vector or example video/frame.
Function: Search task is performed within the local grid and transmitted to the other nodes of the world-wide NewsGrid. Each node generates a list of the best fitting videos. These separate lists are combined to a single one.
Result: Set of results ordered by relevance.

6. Data retrieval using dynamic features (DODV) provided by information supplier: A dynamic content based search is enabled by this service. The search algorithm is made available by the information supplier, e.g. a news agency. As in the case of a combination with static data the search can be limited to a subset of the database.
Input parameters: Algorithm-Id, video object type, example video/frame, algorithm specific data, set of videos to search
Function: Search all video objects of specified type in the subset by comparing their DODV's to the example's (see section 5).
Result: Search job-Id (immediately), set of results ordered by relevance (after search is completed)

7. Data retrieval using user defined dynamic features (DODV): This service is very similar to the previous one with the difference that the algorithm to compare a video with the provided example is defined by the user. Therefore instead of the algorithm-Id an executable is used as input parameter.

8. Monitoring the search progress: Information about the progress of a search in the case of compute intensive content based searches is supplied. It must be possible for the user to ask for the display of intermediate results using the video streaming service.
Input parameters: Search job-Id.
Function: Ask every involved node to estimate the remaining time for the search and calculate overall remaining time.
Result: Overall remaining time.

9. Add new static feature for manual description: This service allows for a new static feature for manual description to be added to all videos in the database.
Input parameters: Name of the feature, data type (using MPEG-7-schema), default value.
Function: A new entry for each video in the database is created and filled with the default value. The schema for new videos is adapted. A message to the archive administrators is created to provide valid entries for the new field.
Result: Success/error.

10. Add new static feature for automatic description: Similar to the last service, but a feature for automatic description generation is added.
Input parameters: Name of the feature, data type (using MPEG-7-schema), algorithm to calculate the feature (executable).
Function: The algorithm to calculate the features is started for every video and the result will be stored in the database.
Result: Success/error

11. Video cutting: This service enables users to extract various cuts from different videos and join these to form a single new video. This is especially then the case when topical videos are to be combined with archived material. The video cutting process can thus be executed directly on the data extracted from the repository. The advantages of this service are twofold. In the first instance the amount of data to be downloaded can be substantially reduced. Secondly, news editors who only

have a moderately powerful computer, such as a notebook, available can execute cutting tasks on a more powerful grid node.
Input parameters: Video-codec, audio-codec, quality/bitrate, transmission standard and for each video to be part of the result: video-Id, start and end-position, fading-technique.
Function: Cut the video segments from the original videos and merge them to one video using fading techniques. Merge SODVs to one single SODV. Generate video-Id for the result video.
Result: Video-Id.

9. Conclusions

In this paper an infrastructure for and services supplied by a world wide NewsGrid is described. The distributed digital archive, supplying versatile services for storing, searching and retrieving, downloading, and post-processing of news videos, change the workflow for news editors. The users of NewsGrid obtain access to videos from all over the world, from any TV station or news agency connected to the grid. The data is transmitted only on demand, which saves bandwidth and storage capacity compared to current procedures.

Future work includes the development of specialised user clients, e.g. for mobile devices. The design of the services will be more detailed and more services will be added to the system. The transferability to other use cases will be improved. Furthermore concepts for digital rights managements and user roles must be defined.

References

[1] S. Geisler, G. Joubert: NewsGrid. L. Grandinetti (Ed.): Grid Computing: The New Frontier of High Performance Computing in Advances in Parallel Processing, Vol. 14, Amsterdam, etc.: Elsevier. to be published 2005 (The paper can be downloaded by the reviewers from www.in.tu-clausthal.de/~sgeisler/newsgrid/paper1.pdf using user name parco2005 and password malaga)

[2] E. Oomoto, K. Tanaka: OVID: Design and Implementation of a Video Object Database System. IEEE Trans. on Knowledge and Data Engineering, 5:629643. 1993

[3] G. Falkemeier, G. Joubert, O. Kao: Internet Supported Analysis and Presentation of MPEG Compressed Newsfeeds. Intern. Journal of Computers and Applications, Vol. 23(2), 129-136, ACTA Press. 2001

[4] S. Geisler: Efficient Parallel Search in Video Databases with Dynamic Feature Extraction, Parallel Computing: Software Technology, Algorithms, Architecture And Applications: Proc. of the Intern. Conference Parco2003, 431-438, 2004.

[5] S. Geisler, O. Kao: Cluster-Based Organisation and Retrieval of Newsfeed Archives. Proc. of the Intern. Workshop on Intelligent Multimedia Computing and Networking (IMMCN'2002), 1033-1036. 2002

[6] S.-F. Chang, W. Chen, H. Meng, H. Sundaram, D. Zhong: VideoQ: An Automated Content Based Video Search System Using Visual Cues. Proc. of ACM Multimedia, 313324. 1997

[7] D. Ponceleon, S. Srinivasan, A. Amir, D. Petkovic: Key to Effective Video Retrieval: Effective Cataloging and Browsing. Proc. of ACM Multimedia, 99107. 1998

[8] O. Kao, S. Stapel: Case study: Cairo a Distributed Image Retrieval System for Cluster Architectures. In T.K. Shih (edt.), Distributed Multimedia Databases: Techniques and Applications, 291303. Idea Group Publishing. 2001

[9] S. Geisler: Content-based Video Coding of Newscasts for Mobile Devices. International Conference on Imaging Science, Systems, and Technology (CISST 2005), 109-115. 2005

Provision of Fault Tolerance with Grid-enabled and SLA-aware Resource Management Systems*

Felix Heine[a] , Matthias Hovestadt[a], Odej Kao[a], Axel Keller[a]

[a]Paderborn Center for Parallel Computing (PC^2), Universität Paderborn, Germany

Future applications of the Next Generation Grid will demand for flexible negotiation mechanisms supporting various ways of Quality-of-Service (QoS) guarantees. In this context a Service Level Agreement (SLA) is a powerful instrument for describing all obligations and expectations within a business partnership. Many research projects already focus on realizing SLAs within the Grid middleware. However, this is not sufficient. Resource Management Systems also have to be SLA-aware, since these systems provide their resources to Grid infrastructures. In this paper we present the EU-funded project HPC4U (Highly Predictable Clusters for Internet Grids), which aims at realizing such an RMS by means of SLA-negotiation and SLA-aware scheduling, and application-transparent fault tolerance.

1. Introduction

Research on Grid computing started under solely technical aspects: how to realize this virtualization of resources, and how these distributed virtual resources can be used. Companies like IBM, Hewlett Packard and Microsoft have recognized the potential of Grid Computing and are investing noticeable efforts on research and the support of research communities. Common goal is to attract commercial users for Grid Computing. In this context, the European Commission convened a group of experts to clarify the demands of future Grid systems and which properties and capabilities are missing in currently existing Grid infrastructures. Their work resulted in the idea of the Next Generation Grid (NGG) [6].

The guaranteed provision of reliability, transparency and Quality of Service (QoS) is an important demand of the NGG. Commercial users will not use a Grid system for computing business critical jobs, if it is operating on the best-effort approach only. The user must be able to rely on getting the requested QoS level. At this, QoS does not only imply predictable operation of a single resource, but also the orchestrated execution of a workflow.

Service Level Agreements (SLAs) are powerful instruments for describing all expectations and obligations in the business relationship between service customer and service provider [2]. Such an SLA specifies the QoS requirement profile of a job. At the layer of Grid middleware many research activities already focus on integrating SLA functionality. However, there is a gap between the requirements of an SLA-aware Grid middleware and the capabilities of currently available Resource Management Systems (RMS), offering only best-effort service. Local resource management systems provide their resources to Grid systems, which transfer jobs from Grid users to these provided resources. Hence, SLAs guaranteed by the Grid middleware must be realized by local RMS [11]. However, if the local RMS operates at best-effort, Grid middleware can not assure specific service levels.

A major research focus at PC^2 is on resource management systems providing an increased level of quality of service as a software-only solution. At this, application transparency is crucial. Arbi-

*This work has been partially supported by the EU within the 6th Framework Programme under contract IST-511531 "Highly Predictable Cluster for Internet-Grids" (HPC4U).

trary applications should benefit without recompilation and linkage against special libraries from an increased level of QoS, like fault tolerant job execution. This is of particular interest for commercial Grid environments, since source code of commercial applications is normally not available.

Within the EU-funded project "Highly Predictable Cluster for Internet-Grids" (HPC4U) [7] the PC^2 is working on an SLA-aware resource management system, utilizing the mechanisms of the process, storage and network subsystems for realizing application-transparent fault tolerance. This paper will first describe the architecture of the HPC4U cluster middleware system, then explaining its mechanisms of QoS provision. The paper will conclude with a short summary and an overview about future work.

2. Related Work

The worldwide research in Grid computing resulted in numerous different Grid packages. Besides many commodity Grid systems, general purpose toolkits exist such as UNICORE [13] or Globus [5]. Although Globus represents the de-facto standard for Grid toolkits, all these systems have proprietary designs and interfaces. To ensure future interoperability of Grid systems as well as the opportunity to customize installations, the OGSA (Open Grid Services Architecture) working group within the GGF [3] aims to develop the architecture for an open Grid infrastructure [4].

In [6], important requirements for the Next Generation Grid (NGG) were described. Among those needs, one of the major goals is to support resource-sharing in virtual organizations all over the world. Thus attracting commercial users to use the Grid, to develop Grid enabled applications, and to offer their resources in the Grid. Mandatory prerequisites are flexibility, transparency, reliability, and the application of SLAs to guarantee a negotiated QoS level.

An architecture that supports the co-allocation of multiple resource types, such as processors and network bandwidth, was presented in [8]. The Globus Architecture for Reservation and Allocation (GARA) provides "wrapper" functions to enhance a local RMS not capable of supporting advance reservations with this functionality. This is an important step towards an integrated QoS aware resource management.

In this paper, this approach is enhanced by SLA and monitoring facilities. These enhancements are needed in order to guarantee the compliance with all accepted SLAs. This means, it has to be ensured that the system works as expected at any time, not only at the time a reservation is made. The GARA component of Globus currently does neither support the definition of SLAs or malleable reservations, nor does it support resilience mechanisms to handle resource outages or failures.

The requirements and procedures of a protocol for negotiating SLAs were described in SNAP [9]. However, the important issue of how to map, implement, and assure those SLAs during the whole lifetime of a request on the RMS layer remains to be solved. This issue is also addressed by the architecture presented in this paper.

The Grid community has identified the need for a standard for SLA description and negotiation. This led to the development of WS-Agreement/-Negotiation [1]. These upcoming standards rely on the new Web Services Resource Framework (WSRF, [10]) which will supersede the Open Grid Services Infrastructure (OGSI, [12]) specification. We will follow these developments closely and will stick to these standards.

3. Architecture of HPC4U

The HPC4U cluster middleware will consist of multiple elements, i. e. the SLA-aware RMS and the main building blocks for ensuring a high level of fault tolerance: process checkpoint, storage

snapshot and virtualization, and network failover. In an exceptional situation, e. g. the outage of hardware resources, the HPC4U system will use its fault tolerance mechanisms to assure the completion of a job. This means that the process checkpointing software enables checkpoint/restart (and migration) of a running process, so that jobs can be restarted from the last checkpoint on a spare resource. But only considering the checkpoint process could cause inconsistencies at restart, because a job continues to write data on files after it has been checkpointed. Hence, the system has to maintain consistency between the checkpointed process and the storage. The checkpointing process also has to be supported by the network subsystem, e.g. regarding in-transit network packets.

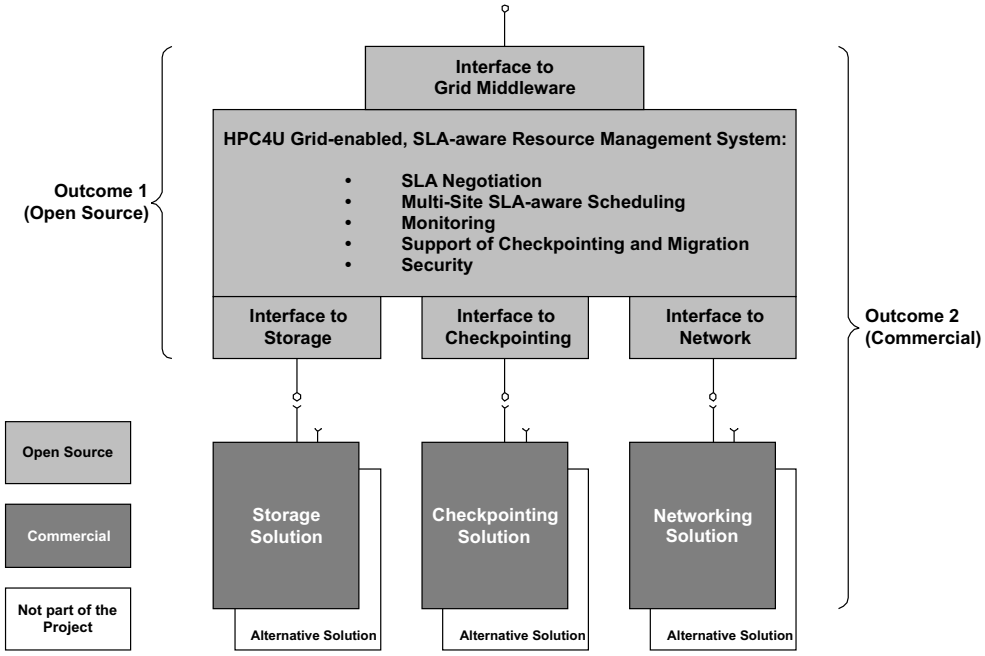


Figure 1. Outcomes of the HPC4U project

The results of HPC4U will be a mix of open source and proprietary software embedded in two outcomes (cf. Figure 1). The SLA-aware and Grid-enabled Resource Management System includes SLA negotiation, multi-site SLA-aware scheduling, security and interfaces for storage, checkpointing, and networking support. It will be multi-platform in nature and available as open source. The second HPC4U outcome will be a vertically integrated commercial product with proprietary Linux-specific developments for storage, networking and checkpointing. This outcome will demonstrate the entire, ready-to-use HPC4U functionality (job checkpointing, migration, and restart) for Grids based on Linux architectures. It is obvious that providing an agreed level of Quality of Service and Fault Tolerance requires broad interaction between all components of the HPC4U system.

Within the HPC4U project a cluster middleware system will be developed, which consists of three independent layers. At the upper level, HPC4U will provide an interface, which can be used by Grid middleware systems to negotiate on Service Level Agreements. To provide maximum flexibility and compatibility with other research projects and software systems, we will closely follow existing standards.

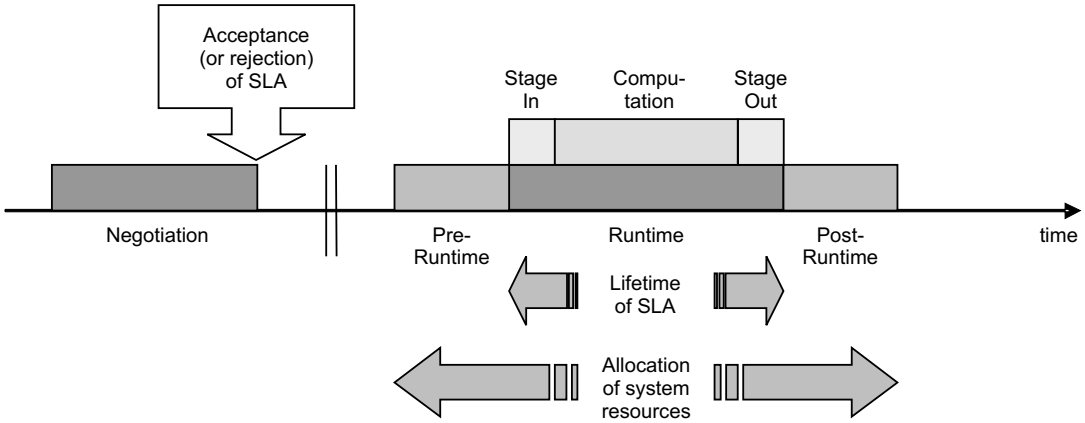


Figure 2. Phases of operation

An SLA-aware resource management system represents the middle layer of the HPC4U cluster middleware architecture. Using the above mentioned interface it negotiates with customers on SLAs. It also assures the compliance with these agreed SLAs at runtime. This does not only imply the monitoring of internal resources, but also the utilization of appropriate mechanisms to realize fault tolerance in case of resource outages.

These mechanisms base on functionalities which will be provided by the subsystems of HPC4U, which represent the lower level in the HPC4U cluster middleware architecture. Offering specific APIs, each of these subsystems provides special mechanisms for fault tolerance on process-, network- or storage-level. Since all interfaces within the HPC4U system are standardized and published, each component can be replaced with arbitrary third-party products, as long as these products provide compliant interfaces.

4. Provision of QoS

Negotiation is the initial step in the SLA lifetime. Here, the service consumer (e. g. end-user in the Grid) and service provider (e. g. an SLA-aware resource management system) first have to agree on the contents of an SLA. If the RMS agrees on the contents of an SLA request at the end of the negotiation process, it is responsible for fulfilling all demands and requirements of this SLA. Figure 2 depicts that the negotiation process may take place arbitrarily long in advance to the actual resource consumption. Between the successful completion of the negotiation process (which results in a new SLA) and the actual resource consumption starting at the pre-runtime phase, the RMS does not have to provide any resources for this SLA. However, it is aware of this SLAs, considering its demands in the scheduling process.

Before resources have to be provided according to the terms of the SLA, the system has to be configured (e. g. initialization of storage and network, or configuration of compute nodes) in the pre-runtime phase. These configured resources will be re-configured to normal operation after the job has been completed. During the runtime phase, the RMS has to ensure the adherence will all terms this SLA. For this, it uses its fault tolerance mechanisms for handling outages like failures of network, storage, or compute nodes.

4.1. Checkpointing of a Job

Resource outages (e. g. power failure of a compute node) usually cause a crash of the job running on these resources. Without checkpointing mechanisms, such a job has to be restarted from the very beginning, so that all computational results are lost. Especially for long running jobs, this is a major drawback. Deadline guarantees for such jobs are at most best effort. If a weather service uses Grid resources for computing the weekend weather forecast, the computed result would be useless if it is finished on Monday due to resource outages.

With system level checkpointing mechanisms available, the resource management system is able to create images of a running process in regular intervals. In case of resource outages, the resource management system can query for compatible and suitable spare resources to resume the job. If such resources were found, the checkpoint dataset (e. g. the process image) is transferred to the new compute node. There the job can restart from the last checkpoint. The impact on the time of completion of the job is minimal, since it is only delayed for the time required for checkpointing, dataset transfer, and restart. However, the job does not have to be restarted completely. This leverages the realization of fault tolerance and the provision of deadline guarantees. The HPC4U project will realize an application-transparent checkpointing, so that arbitrary applications can benefit from this service.

However, it is not sufficient to focus on process checkpointing only. The storage subsystem will provide checkpointing mechanisms for the storage partition of a running job. If the resource management system initiates a checkpoint of a running process, it simultaneously starts the checkpoint of the data partition. If the job needs to be restarted due to a resource failure at a later time, both process and storage are restored. This assures the consistency of the running process with its saved data.

If an application is running on multiple nodes in parallel, each node may send messages to the other nodes of the application (e. g. information exchange or synchronization). If such an application is checkpointed, a node might currently be sending a message to another node. At time of checkpoint, this message might have already left the sender, but not yet reached the recipient. Such a packed is called in-transit. Consistency is a major demand to the checkpointing mechanism. Hence, the network subsystem must handle these in-transit packets. At checkpoint time, the checkpointing subsystem will first freeze the application, so that its state is stable and does not change. Now the network subsystem is invoked to check all stacks for network packets. Also the network itself is checked for currently transmitted packets. All these packets are saved in a network dataset file.

4.2. Migration of a Job

To resume the computation of a job in case of a resource outage, the RMS first has to locate suitable free spare resources. For this query process, it can contact two information sources. First, the RMS knows about the SLA of the running job. This SLA already contains a detailed requirement profile of the job, which can be used for finding suitable resources.

But solely focusing on the SLA of a running application is not yet sufficient. The process subsystem of HPC4U will provide a mechanism called ”virtual bubble”. This bubble virtualizes the resources of a compute node. Applications which have been started inside such a bubble are only aware of these virtual resources, e. g. virtual network devices. By checkpointing the entire bubble, arbitrary applications can be checkpointed without any need for recompilation or relinking.

By checkpointing a running process within a virtual bubble, many dependencies between the application and its environment arise. These range from the type of the currently used processor, the operating system, and the exact kernel version, up to the location and exact version of libraries. The RMS needs also to cope with these constraints for finding really compatible spare resources. There-

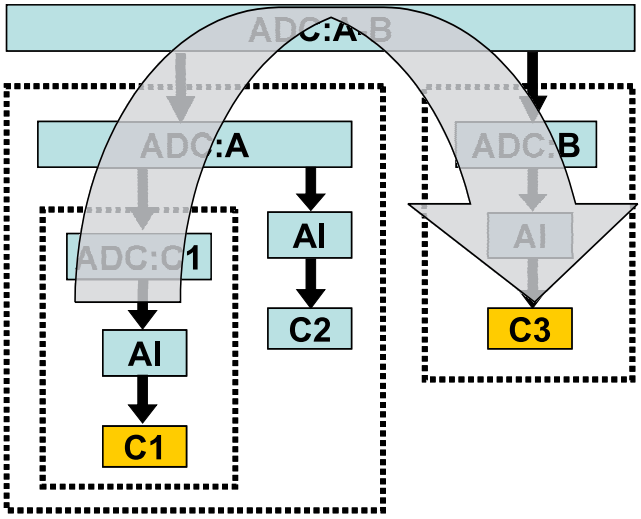


Figure 3. Migration to different cluster system

fore it first invokes the API of the checkpointing subsystem, querying for a compatibility profile of the checkpoint dataset. The checkpoint subsystem now starts to analyse the checkpoint dataset and returns and XML file, containing the demanded profile.

Based on these two sources of information, the RMS starts searching appropriate spare resources. At this, it tries to solve the problem as local as possible. This means, before migrating jobs to Grid resources, it first tries to find resources within the same cluster system or within the same administrative domain. Even if this strategy is not fixed and can be modified, it is preferable as the price of a migration process normally increases with the distance (i. e. the bandwidth capacity of the network between source system and target of migration).

Depending on the location of the found spare resource (local, administrative domain, Grid), the RMS invokes specific migration mechanisms. This implies the task of making the checkpoint dataset of the affected application available on the target resource.

If all preparatory tasks are completed, the RMS of the target resource invokes the API of the checkpointing subsystem to resume the job from the checkpoint dataset. The job will again run in a virtual bubble.

4.3. Example

In the HPC4U system, an Administrative Domain Controller (ADC) is responsible for establishing an administrative domain, where a given set of policies regarding security, access, accounting, or runtime responsibility is valid. The ADC serves as a central gateway between the internal resources of the administrative domain. Each of these internal resources (e. g. several cluster systems) is operated by means of a local resource management systems. To unify the interface between the ADC and the specific internal RMS, the ADC does not interface the RMS directly. Instead, an Active Interface (AI) acts as an adapter between the demands of the ADC and the capabilities of the specific internal RMS.

In the example scenario depicted in figure 3, cluster `C1` is operated by a subdepartment of a large company. This subdepartment has its specific policies on resource usage. Hence, the administrative domain controller `ADC:C1` is in charge of enforcing these policies. The subdepartment belongs to a department, again forming an administrative domain. This department is also operating cluster

system C2. The company also has a second department, operating cluster system C3. The company itself forms the surrounding administrative domain.

As explained above, cluster C1 is in charge of fulfilling all accepted SLAs. Assuming a resource outage at C1, C1 first tries to handle the problem internally, e. g. by rescheduling the jobs according the SLAs and resuming the job from the last checkpointed state. In case the RMS cannot realize the adherence with all SLAs (e. g. the resource outage has affected to much compute nodes), it tries to request for spare resources at the next higher level in hierarchy. ADC:A now tries to handle the problem internally, i. e. negotiating with cluster C2 on completing the job. If C2 is not able to accept the job (e. g. due to incompatible resources or high system utilization), ADC:A forwards the request to ADC:A-B, which is now trying to handle the problem internally. If cluster C3 accepts the SLA-request of C1, a migration process of all checkpoint data is initiated.

5. Conclusion and Future Work

In this paper we have outlined the basic ideas and components of the HPC4U cluster middleware system. HPC4U's main components are the SLA-aware resource management system and the subsystems for realizing fault tolerance on process, storage, and network.

The goal of the HPC4U project is to provide an application-transparent and software-only solution of a reliable Resource Management System. It will allow the Grid user to negotiate on Service Level Agreements, which will be realized by means of process and storage checkpointing, and other sophisticated mechanisms. By this, the HPC4U cluster middleware will be an important building block for realizing Next Generation Grids.

Application-transparency means that applications will not notice that they are running in an HPC4U environment, since they are running in the virtualized environment of a "virtual bubble". The application does not have to be modified, recompiled, or relinked in any way to benefit from the HPC4U fault tolerance mechanisms. Hence, the HPC4U system is able to provide checkpoint and migration service also for commercial applications where normally no source code is available.

The HPC4U solution will not only passively accept resource requests from Grid users, it will also act as an active Grid component. If the HPC4U system can not compensate resource outages, so that the fulfillment of agreed SLAs is endangered, it may request the Grid for suitable spare resources. If such resources are found, the job will be transparently migrated. This way, available Grid resources are used for further improving the level of Fault Tolerance.

Currently the HPC4U project is within the second of four technological workpackages. This workpackage addresses the first of three major steps in building up this system, namely the realization of the needed fault tolerance extensions to storage, communication, and system software in a single node environment. The development and implementation of these basic mechanisms serve as a fundament for merging single nodes into an Intranet Grid and then for including the Intranet Grid into the world wide Grids. The core tasks in this workpackage are related to preparing the building blocks for a grid-wide job migration and have the main goal, to integrate the job checkpointing with the storage and resource management component. Furthermore, a RMS monitoring mechanism will collect information about the available resources and their status.

The realization of this vertical approach is based on existing software solutions of the HPC4U partners. A first prototype implementation of our architecture has already been finished. It enables the user to request for a fault tolerant handling of his single-node jobs. The HPC4U system starts such a job within a virtual bubble, using the subsystems for transparent checkpoint and migration within the same cluster system. Ongoing work within HPC4U focuses on providing checkpointing and migration also to parallel-node jobs, and the realization of inter-cluster Grid migration.

Workpackage 2 will be finalized in mid 2005. Within the scope of the succeeding workpackage the existing FT mechanisms of the HPC4U system will be extended to multi-node/Intranet Grid environments on the one hand and to distributed running multi-node jobs on the other hand. The extension to Intranet Grids means, that the RMS must find suitable resources within this domain as a target for the job migration. This includes the suitability of the software and hardware architecture, the availability of the required resources and the compliance with the existing SLAs. Thus, for each migrated job, a start time for resumed processing will be assigned in a way that the given deadline can be reached, if the process duration information supplied by the user is correct.

The extension to multi-node jobs is a large scientific challenge, as already existing mechanisms are limited to single-node jobs. The migration of multi-node jobs affects the checkpointing, migration and restart mechanisms on job-, storage-, and communication-level, which must be able to deal with the specific characteristics of a multi-node job. The RMS has to be capable of handling multi-node jobs, since new requirements arise for compatibility, portability, and migration.

Moreover, the HPC4U system resulting of this workpackage will be capable of cross-border migration, allowing an RMS to migrate jobs on resources within the own administrative domain or over multiple administrative domains. This will further increase the Fault Tolerance, as (temporarily) HW/SW/Network failures can be compensated with a higher probability, as the pool of appropriate resources is significantly enlarged by considering all cross-border resources.

References

[1] A. Andrieux et al. Web Services Agreement Specification (WS-Agreement). `http://www.gridforum.org/Meetings/GGF11/Documents/draft-ggf-graap-agreement.pdf`, 2004.

[2] A. Sahai et al. Specifying and Monitoring Guarantees in Commercial Grids through SLA. Technical Report HPL-2002-324, Internet Systems and Storage Laboratory, HP Laboratories Palo Alto, November 2002.

[3] Global Grid Forum. `http://www.ggf.org`.

[4] GGF Open Grid Services Architecture Working Group (OGSA WG). Open Grid Services Architecture: A Roadmap, April 2003.

[5] Globus Alliance: Globus Toolkit. `http://www.globus.org`.

[6] H. Bal et al. Next Generation Grids 2: Requirements and Options for European Grids Research 2005-2010 and Beyond. `ftp://ftp.cordis.lu/pub/ist/docs/ngg2_eg_final.pdf`, 2004.

[7] Highly Predictable Cluster for Internet-Grids (HPC4U), EU-funded project IST-511531. `http://www.hpc4u.org`.

[8] I. Foster et al. A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation. In *7th International Workshop on Quality of Service (IWQoS), London, UK*, 1999.

[9] K. Czajkowski et al. SNAP: A Protocol for Negotiating Service Level Agreements and Coordinating Resource Management in Distributed Systems. In U. Schwiegelshohn (Eds.) D.G. Feitelson, L. Rudolph, editor, *Job Scheduling Strategies for Parallel Processing, 8th InternationalWorkshop, Edinburgh,*, 2002.

[10] Karl Czajkowski et al. The WS-Resource Framework. `http://www.globus.org/wsrf/specs/ws-wsrf.pdf`, 2004.

[11] L.-O. Burchard et al. The Virtual Resource Manager: An Architecture for SLA-aware Resource Management. In *4th Intl. IEEE/ACM Intl. Symposium on Cluster Computing and the Grid (CCGrid) Chicago, USA*, 2004.

[12] S. Tuecke et al. Open Grid Services Infrastructure (OGSI) V1.0. `http://forge.gridforum.org/projects/ggf-editor/document/draft-ogsi-service-1/en/1`, 2003.

[13] UNICORE Forum e.V. `http://www.unicore.org`.

Air pollution forecast on the HUNGRID infrastructure

R. Lovas[a], J. Patvarczki[a], P. Kacsuk[a], I. Lagzi[b], T. Turányi[b], L. Kullmann[c], L. Haszpra[c], R. Mészáros[d], A. Horányi[c], Á. Bencsura[e], Gy. Lendvay[e]

[a]Computer and Automation Research Institute, Hungarian Academy of Sciences (MTA SZTAKI), Budapest, Hungary

[b]Department of Physical Chemistry, Institute of Chemistry, Eötvös University, Budapest, Hungary

[c]Hungarian Meteorological Service, Budapest, Hungary

[d]Department of Meteorology, Eötvös University, Budapest, Hungary

[e]Institute of Chemistry, Chemical Research Center of the Hungarian Academy of Sciences, Budapest, Hungary

Computational Grid systems are gaining more and more attention in the natural sciences but very often the end-users (biologists, chemists, physics) must tackle various problems when they want to deploy such systems. In this paper a unified software development family is presented, which is able to cover each stage of parallel software development as well as the seamless application migration from parallel systems to Grid platforms. Besides the recently established HUNGRID infrastructure, the development life-cycle is also presented through two air-pollution modelling applications, which enable the authorities to prevent the harmful effects of high-level ozone concentration and accidental releases. The developed computational models can play crucial role in the management of the photochemical smog episodes; they can be used to test the effects of the ozone fluxes and possible emission control strategies and accidents.

1. Introduction

Computational Grid systems [1] are gaining more and more attention in the natural sciences. In such systems, a large number of heterogeneous computer resources are interconnected to solve complex problems. The main aim of the national research project, 'Chemistry Grid and its application for air pollution forecast' [2], was the investigation of the feasible applications of Grid technology in computational chemistry from practical aspects; e.g. prevention of the harmful effects of high-level ozone concentration. The project relied on an academic product family of MTA SZTAKI; a Grid monitoring tool, called Mercury [3], and two integrated application development environments, called P-GRADE parallel programming environment [4] (see Figure 2), and P-GRADE Grid portal [5] (see Figure 3). These tools enable the parallelisation and 'gridification' of sequential applications in a more flexible and transparent way than other solutions [2,6,7] by means of their high level graphical approach, multi-grid support, performance analyzer and debugging tools. Recently, the P-GRADE portal has been developed further to provide support for the efficient execution of complex programs in various Grids. It includes the dynamic execution of applications across the Grid resources according to the actual state and availability conditions provided by the new information/broker system. In the project, the consortium applied these achievements for supporting a specific e-Science area; the established infrastructure can provide access for chemists to Hungarian computational Grid resources, called HUNGRID, which is not only a virtual organization within the EGEE project [8] but it contains new elements, such as P-GRADE portal [5] and MERCURY [3],

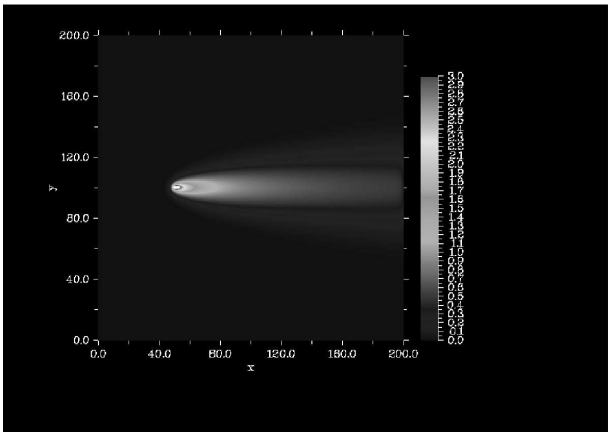


Figure 1. Result of simulation: distribution of chemical (radioactive) substances.

making easier the use of the infrastructure for solving complex problems, such as modelling of air pollution.

2. Application I: Accidental release of chemical substances

Modelling the accidental release of chemical (or radioactive) substances from a single source requires that the numerical simulations must be achieved obviously faster than in real case in order to use them in decision support. A feasible way is the parallelization of source code. Evolution of chemical species can be described by second-order partial differential equations:

$$\frac{\partial c_i}{\partial t} = K_{x,i}\frac{\partial^2 c_i}{\partial x^2} + K_{y,i}\frac{\partial^2 c_i}{\partial y^2} - u\frac{\partial c_i}{\partial x} - v\frac{\partial c_i}{\partial y} + R_i(c_1, c_2, \ldots, c_n), \quad i = 1, 2, \ldots, n, \tag{1}$$

where c_i is the concentration, $K_{x,i}$, $K_{y,i}$ are the turbulent diffusion coefficients, u, v are the components of the horizontal wind velocity and R_i is the chemical reaction term, respectively, of the ith chemical species. t is time, and x and y are the spatial variables. The chemical reaction term R_i may contain non-linear terms in c_i. For n chemical species, an n dimensional set of partial differential equations is formed describing the change of concentrations over time and space. These equations are coupled through the non-linear chemical reaction term.
The basis of the numerical method for the solution of the partial differential equations is the spatial discretisation of the partial differential equations on a two-dimensional rectangular grid. In these calculations, the grid spacing is uniform in both spatial directions. This approach, known as the 'method of lines', reduces the set of partial differential equations (PDEs) of three independent variables (x, y, t) to a system of ordinary differential equations (ODEs) of one independent variable, time. A second order Runge-Kutta method is used to solve the system of ODEs arising from the discretisation of the transport terms with chemistry.

2.1. Parallel implementation in P-GRADE

The graphical language of P-GRADE consists of three hierarchical design layers: (i) Application Layer is a graphical level, which is used to define the component processes, their communication ports as well as their connecting communication channels. Shortly, the Application Layer serves for describing the interconnection topology of the component processes or process groups (see Figure 2,

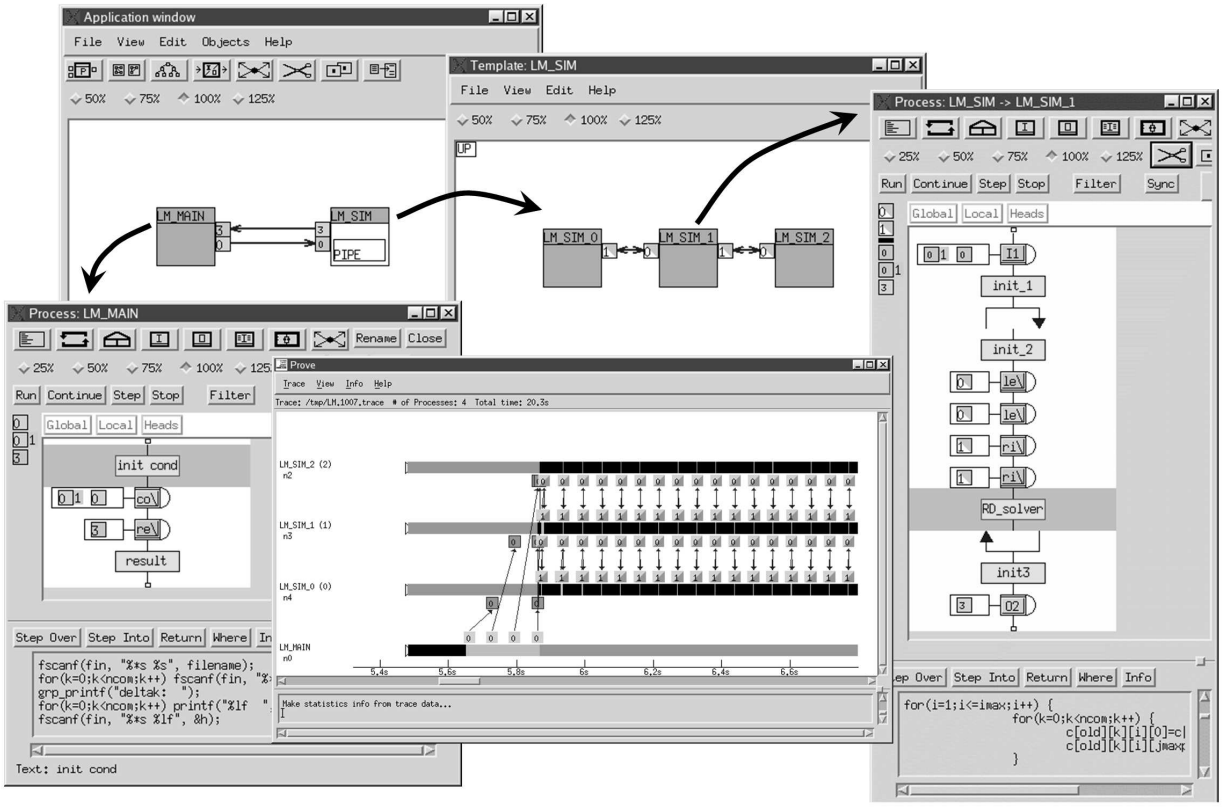


Figure 2. Accidental release simulation: parallel code in P-GRADE development environment.

Application window). (ii) Process Layer is also a graphical level where different types of graphical blocks are applied: loop construct (see Figure 2, blue arcs in the window labelled Process: *LM_SIM* → *LM_SIM_1*), conditional construct, sequential block, input/output activity block and macrograph block. The graphical blocks can be arranged in a flowchart-like graph to describe the internal structure (i.e. the behaviour) of individual processes (see Figure 2, Process windows). (iii) Text Layer is used to define those parts of the program that are inherently sequential and hence only pure textual languages like C/C++ or FORTRAN can be applied at the lowest design level. These textual codes are defined inside the sequential blocks of the Process layer (see Figure 1, at bottom of Process window labelled Process: *LM_SIM* → *LM_SIM_1*).

In order to parallelise the sequential code of the presented accidental release simulation the domain decomposition concept was followed; the two-dimensional grid is partitioned along the x space direction, so the domain is decomposed into horizontal columns. Therefore, the two-dimensional subdomains can be mapped onto e.g. a pipe of processes (see Figure 2, Template: *LM_SIM* window). An equal partition of sub-domains among the processes gives us a well balanced load during the solution of the reaction-diffusion-advection equations assuming a homogeneous cluster (see Figure 2, PROVE performance visualisation window) or a dedicated supercomputer as the execution platform. During the calculation of the diffusion of the chemical species communications are required to exchange information on the boundary concentrations between the nearest neighbour sub-domains, which are implemented via communication ports, channels (see Figure 2, Template: *LM_SIM* window, arcs between small rectangles), and communication actions (see Figure 2, Process: *LM_SIM* → *LM_SIM_1*, icons labelled as '*le*' and '*ri*' in the control flow like description). For the calculation the

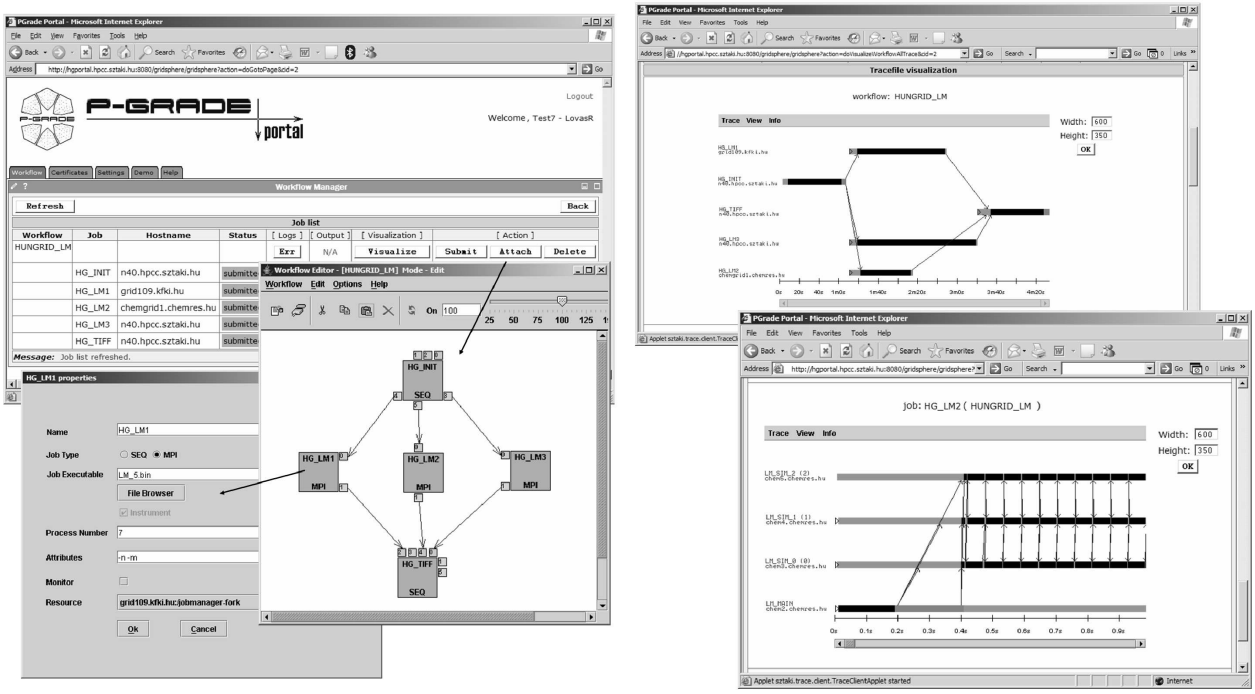


Figure 3. Accidental release simulation on HUNGRID: workflow and job descriptions in P-GRADE portal and execution visualisation at both levels.

process invokes sequential code segments (see Figure 2, bottom of Process: *LM_SIM* → *LM_SIM_1* windows). Based on the hierarchical graphical description and the given sequential code segments, we generated automatically the instrumented MPICH code with the P-GRADE environment. In the performance analysis phase, the underlying MERCURY monitor [3] collects the trace information for the on-line performance visualization. For illustration purposes, in Figure 2 the PROVE window depicts the space-time diagram of the execution with 4 processes, where the user can inspect the behaviour of the application; e.g. the initialization phase, the multicast of input parameters (blue arcs between the process bars), or the periodic exchanges of boundary conditions between the processes of pipe communication template. The black colour in process bars represents the periods when the process executes the sequential code segments of calculation, so it can be easily recognized that the simulation can perform well. The parallel version of reaction-diffusion-advection simulation has been tested and fine tuned similarly to the earlier developed chemical simulations [9] on SZTAKI cluster (a part of HUNGRID infrastructure) using it as a dedicated resource. This self-made Linux cluster contains 29 dual-processor nodes (Pentium III/500MHz) connected via Fast Ethernet.

2.2. Workflow description and execution on HUNGRID with P-GRADE portal

P-GRADE portal [5] is a workflow-oriented Grid portal with the main goal to enable users to manage the whole lifecycle of workflow-oriented complex grid applications. The P-GRADE portal supports the graphical development of workflows (see Figure 3, Workflow editor) created from various types of existing components (sequential, MPI or PVM jobs), executing these job-workflows in the Grid relying on user credentials, and finally analyzing the monitored trace-data by the built-in visualization facilities. The P-GRADE Portal provides the following functions: Creation and modification of workflow applications; Setting the Grid environment; Managing Grid certificates;

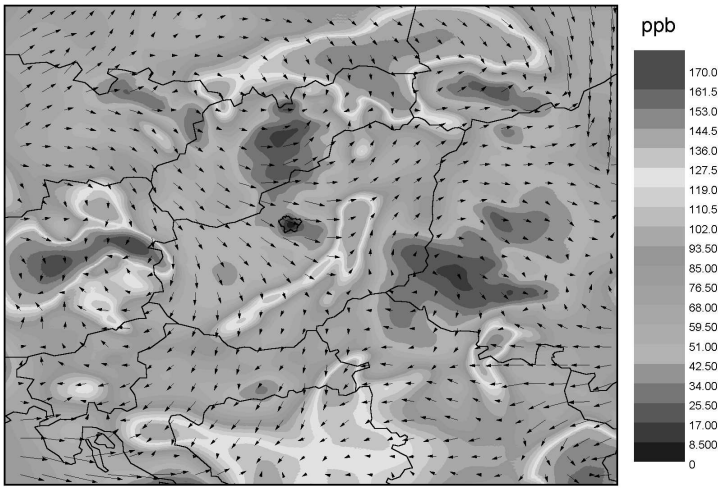


Figure 4. Calculated ozone concentrations on the 3rd of August, 1998 at 17.00 with wind field originated from the ALADIN weather prediction model.

Managing the execution of workflow applications on grid resources; Visualizing the progress of workflows and their component jobs (see Figure 3).
The simulation was executed on the HUNGRID infrastructure, which is the Hungarian Virtual Organisation (VO) of the EGEE Grid [8] based on LCG-2 with some extensions developed as part of the Hungarian Grid research efforts. HUNGRID was created in the framework of the Hungarian Grid project funded by IHM and has got about 500 machines from 5 sites and two other sites have already decided to join with another 150 machines.
The presented workflow version of accidental release simulation contains 5 jobs. The first sequential job, HG_INIT (see Figure 3) initialize the input parameters for the different simulation scenarios with varying emission factors, wind speed, altitude of the source, time limit, etc. Then, three parallel jobs, HG_LM1 HG_LM3 calculate the effects of the different accidental release scenarios. These parallel jobs were developed by the P-GRADE development environment (see Section 2.1), and the instrumented MPICH executables were generated by the P-GRADE run-time system. Finally, the last sequential job, HG_TIFF is responsible for the creation of emission diagrams based on the calculated results (see Figure 1).
The execution time was reduced radically, since each simulation job was executed on different sites of HUNGRID VO in parallel (see Figure 3, top window in right), the simulation jobs were parallel applications itself (see Figure 3, bottom window in right), and the communication and initialization overhead was quite acceptable as they can be seen in Figure 3. In the frame of SEEGRID projects, the accidental release simulation with 8 parallel jobs has been demonstrated successfully with 8 different sites collected from South-European countries at MIPRO conference [10].

3. Application II: Photochemical air pollution

The phytotoxic nature of ozone was recognized decades ago. Due to high emissions of ozone precursor substances, elevated ozone concentrations may cover large areas of Europe for shorter (episodic) or longer periods under certain meteorological conditions. These elevated concentrations

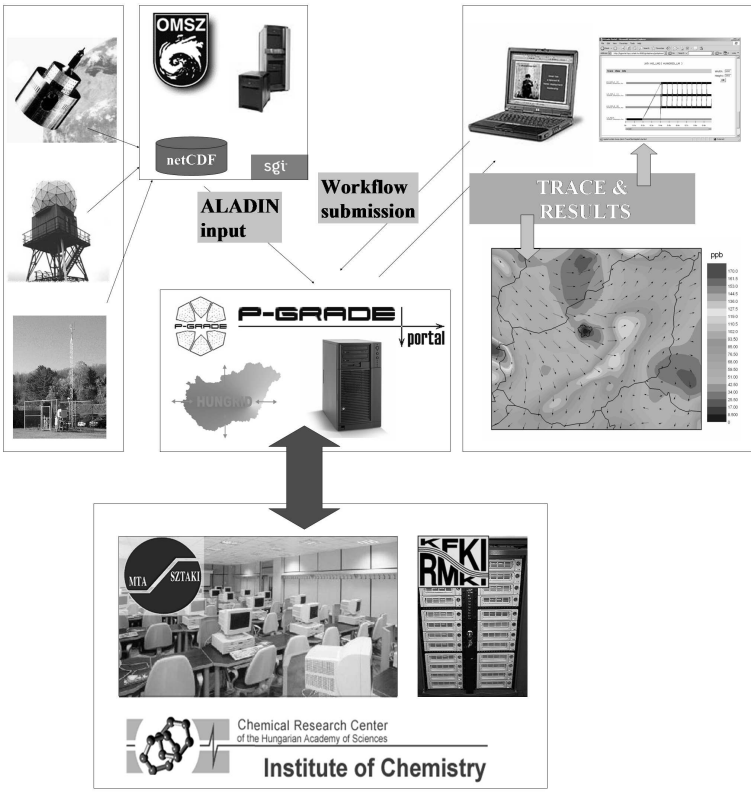


Figure 5. Air pollution forecast on the HUNGRID infrastructure.

can be potentially damaging to agricultural and natural vegetation. Occasional extreme concentrations may cause visible injury to vegetation, while long-term exposure, averaged over the growing season, can result in decreased productivity and crop yield. For the computational study of this phenomenon in Hungary, a coupled Eulerian photochemical reaction-transport model and a detailed ozone dry-deposition model were developed. The Eulerian air pollution model was developed through a co-operation between the Eötvös University, Budapest, The University of Leeds and the Hungarian Meteorological Service [11–13].

This model fully utilized the experience collected previously at the Leeds University on the use of adaptive gridding methods for modelling chemical transport from multi-scale sources. The model has been elaborated within a flexible framework where both area and point pollution sources can be taken into account, and the chemical transformations can be described by a mechanism of arbitrary complexity. The reaction-diffusion-advection equations relating to air pollution formation, transport and deposition are solved on an unstructured triangular grid. The model domain covers Central Europe including Hungary, which is located at the centre of the domain and is covered by a high-resolution nested grid. The sophisticated dry-deposition model estimates the dry-deposition velocity of ozone by calculating the aerodynamics, the quasi-laminar boundary layer and the canopy resistance. The meteorological data utilized in the model were generated by the ALADIN meso-scale limited-area numerical weather prediction model (see Figure 5), which is used by the Hungarian Meteorological Service [14]. For Budapest, the emission inventories for CO, NOx and VOCs were provided by the local authorities with a spatial resolution of 1 km 1 km and also include the most significant 63 emission point sources. For Hungary, the National Emission Inventory of spatial res-

olution 20 km 20 km was applied which included both area and point sources. Outside Hungary, the emission inventory of EMEP for CO, NOx and VOCs was used, having a spatial resolution of 50 km 50 km.
The work demonstrates that the spatial distribution of ozone concentrations is a less accurate measure of the effective ozone load than the spatial distribution of ozone fluxes. The fluxes obtained show characteristic spatial patterns, which depend on soil moisture, meteorological conditions, ozone concentrations and the underlying land use. The simulation of photochemical air pollution is based on the presented approach (see Section 2.1) as well as the experiences concerning the earlier developed P-GRADE version of ultra-short range weather prediction system [10]. The major difference is that their simulation jobs have not been parallelized due to the unavailable source code of some third-party modules in the model. Thus, we were not able to take advantages of multi-level parallelism, only the workflow level, which looks similar to the presented case (see Section 2.1). As it is depicted in Figure 5, the P-GRADE portal server is in the centre of air pollution simulation and dedicated to HUNGRID infrastructure. Currently, it provides access to three clusters located at different academic institutes; MTA SZTAKI, KFKI-RMKI, and CRC-HAS. The portal server has access to the meteorological data (ALADIN input files) as well, which are calculated numerically by the Hungarian Meteorological Service based on the available radar and satellite images, the observations (see Figure 5, left side), and results of other models. The portal server can be accessed remotely by submitting the simulations, i.e. the P-GRADE workflows, and by downloading the visualization of execution traces (see Figure 3, right side) and simulation results (see Figure 4) on the local machine.

4. Summary and related works

Several institutes apply clusters for environmental modelling (e.g. [15]), and some of them have already recognised the advantages of Grid technology (e.g [16]). There are several available workflow managers and Grid portals for e-Science [17,7,16] but the presented P-GRADE development environment together with the P-GRADE Grid portal provide one of the most flexible and unified ways for the parallel application development and deployment on various Grids. In this paper, we demonstrated that the HUNGRID platform with the presented development tools can provide efficient user support for complex applications, e.g. for air pollution forecasting. On the other hand, P-GRADE portal v2.1 is already working as service for HUNGRID (operated by SZTAKI), SEE-Grid (operated by SZTAKI) [10], and the UK National Grid Service (operated by University of Westminster) [18]. Recently, the developer alliance decided to start supporting the Hungarian ClusterGrid [19], the Croatian Grid and the Turkish Grid as well. Moreover, P-GRADE portal is already connected to the EU GridLab testbed [20], and the UK OGSA testbed [21] for demonstration purposes. Therefore, the P-GRADE portal (as one of its most outstanding features) is able to provide seamless multi-grid access to the end-users, and the application developers do not have to tackle the variety of emerging grid technologies and standards, such as Condor, Globus Toolkit, OGSA, GAT, LCG or gLite.

Acknowledgement

The research described in this paper has been supported by the following projects and grants: Hungarian IHM 4671/1/2003 project, Hungarian OTKA T042459, T043770, and D048673 grants, OTKA Instrument Grant M042110, Hungarian IKTA OMFB-00580/2003, GVOP-3.1.1-2004-05-0359/3.0, and EU INFSO-RI-508833 projects.

References

[1] I. Foster, C. Kesselman: Computational Grids, Chapter 2 of The Grid: Blueprint for a New Computing Infrastructure. Morgan-Kaufman. 1999.

[2] R. Lovas, I. Lagzi, L. Kullmann, Á. Bencsura: Chemistry GRID and its Applications for Air Pollution Forecasting. ERCIM News, 61, 18-19, 2005.

[3] N. Podhorszki, Z. Balaton, G. Gombás: Monitoring Message-Passing Parallel Applications in the Grid with GRM and Mercury Monitor, 2nd European Across Grids Conference, AxGrids 2004, Nicosia, Cyprus, 179-181, 2004.

[4] P. Kacsuk, et al.: PGRADE: a Grid Programming Environment. Journal of Grid Computing, 1, 171-197, 2003.

[5] Cs. Németh, G. Dózsa, R. Lovas, P. Kacsuk: The P-GRADE Grid portal. Computational Science and Its Applications - ICCSA 2004: International Conference, Assisi, Italy, Lecture Notes in Computer Science, 3044, 10-19, Springer-Verlag, 2004.

[6] O. Gervasi, A. Lagana: SIMBEX: a portal for the a priori simulation of crossed beam experiments, Future Generation Computer Systems. The International Journal of Grid Computing: Theory, Methods and Applications, 20, 703-715, 2004.

[7] I. Taylor, I. Wang, M. Shields, S. Majithia: Distributed computing with Triana on the Grid. Concurrency and Computation: Practice and Experience, 17, 1-18, 2005.

[8] http://www.eu-egee.org

[9] R. Lovas, P. Kacsuk, I. Lagzi, T. Turányi: Unified development solution for cluster and grid computing and its application in chemistry. Computational Science and Its Applications - ICCSA 2004: International Conference, Assisi, Italy, Lecture Notes in Computer Science, 3044, 226-235, Springer-Verlag, 2004.

[10] J. Patvarczki, et al.: Using the P-GRADE Grid Portal in the SEE-GRID project. 28th International Convention Mipro, Opatija, Adriatic Coast, Croatia 2005.

[11] A. Tomlin, et al.: On the use adaptive gridding methods for modelling chemical transport from multi-scale sources, Atmospheric Environment, 31, 2945-2959, 1997.

[12] I. Lagzi, et al.: The simulation of photochemical smog episodes in Hungary and Central Europe using adaptive gridding models, Lecture Notes in Computer Science, 2074, 67-77, 2001

[13] I. Lagzi, et al.: Modelling ozone fluxes over Hungary, Atmospheric Environment, 38, 6211-6222, 2004

[14] A. Horányi, I. Ihász, G. Radnóti: ARPEGE/ALADIN: A numerical weather prediction model for Central-Europe with the participation of the Hungarian Meteorological Service. Idöjárás, 100, 277-301, 1996.

[15] J. L. Palau, G. Pérez-Landa, M.M. Millán: Air pollution modelling in complex terrain: 'Els Ports-Maestrat' regional-scale study. ERCIM News, 61, 26-27, 2005.

[16] L. Fusco, J. van Bemmelen: Earth observation archives in digital library and grid infrastructures. Data Science Journal, 3, 222-226, 2004.

[17] M. Vanneschi: The programming model of ASSIST, an environment for parallel and distributed portable applications. Parallel Computing, 28, 1709-1732, 2002.

[18] A. Goyeneche, et al.: Experiences with Deploying Legacy Code Applications as Grid Services Using GEMLCA, Proc. of European Grid Conference 2005, Lecture Notes in Computer Science, 3470, Springer, Amsterdam, 851-860, 2005

[19] G. Sipos et al.: Connecting ClusterGrid and P-GRADE Portal. NETWORKSHOP 2005 Conference, Szeged, Hungary, 2005.

[20] http://www.gridlab.org

[21] T. Kiss, et al: Legacy Code Support for Production Grids, Grid 2005 - 6th IEEE/ACM International Workshop on Grid Computing, Seattle, Washington, USA, 2005. (to appear)

Distributed Shared Memory in a Grid Environment

John P Ryan [a], Brian A Coghlan[a],
{john.p.ryan, coghlan}@cs.tcd.ie

[a]Computer Architecture Group, Dept. of Computer Science, Trinity College Dublin, Dublin 2, Ireland.

1. Abstract

Software distributed shared memory (DSM) aims to provide the illusion of a shared memory environment when physical resources do not allow for it. Here we will apply this execution model to the Grid. Typically a DSM runtime incurs substantial overheads that result in severe degradation in performance of an application with respect to a more efficient message passing implementation. We examine mechanisms that have the potential to increase DSM performance by minimizing high-latency inter-process messages and data transfers. Relaxed consistency models are investigated, as well as the use of a grid information system to ascertain topology information. The latter allows for hierarchy-aware management of shared data and synchronization variables. The process of incremental hybridization, where more efficient message-passing mechanisms can incrementally replace those DSM actions that adversely effect performance, is also explored.

2. Introduction

The message passing programming paradigm enables the construction of parallel applications while minimizing the impact of distributing an algorithm across multiple processes by providing simple mechanisms to transfer shared application data between the processes. However, considerable burden is placed on the programmer whereby send/receive message pairs must be explicitly declared, and this can often be a source of errors. Implementations of message passing paradigms currently exist for grid platforms [11].

The shared memory paradigm is a simpler paradigm for constructing parallel applications, as it offers uniform access methods to memory for all user threads of execution, removing the responsibility of explicitly instrumenting the code with data transfer routines, and hence offers a less burdensome method to construct the applications. Its primary disadvantage is its limited scalability; nonetheless, a great deal of parallel software has been written in this manner. A secondary disadvantage has previously been the lack of a common programming interface, but this has been mitigated by the introduction of emerging standards in this area, such as OpenMP [1] . With OpenMP, compiler directives are used to parallelize serial code by explicitly identifying the areas of code that can be executed concurrently. That this parallelization can be done in an incremental fashion is an important feature in promoting the adoption of this standard.

These are the two predominant models for parallel computing. There are implementations of both paradigms for different architectures and platforms. Shared memory programming has the easier programming semantics, while message passing is more efficient and scalable (communication is explicitly defined and overheads such as control messages can be reduced dramatically or even eliminated). Previous work examined approaches to combine the message passing and shared-memory paradigms in order to leverage the benefits of both approaches, especially when the applications are executed in an environment such as a cluster of SMPs [19].

In contrast, Distributed Shared Memory (DSM) implementations aim to provide an abstraction of shared memory to parallel applications executing on ensembles of physically distributed machines. The application developer therefore leverages the benefits of developing in a style similar to shared memory, while harnessing the price/performance benefits associated with distributed memory platforms. Throughout the 1990's there were numerous research projects in the area of Software-only Distributed Shared Memory (S-DSM), e.g. Midway [6], Treadmarks [13], and Brazos [19]. However, little success has been achieved due to poor scalability and the lack of a common Application Programming Interface (API) [7].
The premise of grid computing is that distributed sites make available resources for use by remote users. A grid job may be run on one or more sites, and each site may consist of a heterogeneous group of machines. As Grids are composed of geographically distributed memory machines, traditional shared memory programs may not execute on multiple processors across multiple grid sites. DSM offers a potential solution, but numerous barriers exist to an efficient implementation on the Grid. However, if the paradigm could be made available, then according to [9], grid programming would be reduced to optimizing the assignment and use of threads and the communication system.
Our aim has been to explore a composition of the environments, with an OpenMP-compatible DSM system layered on top of a version of the now ubiquitous Message Passing Interface (MPI) standard, that can execute in a Grid environment. This choice reflects a desire for a tight integration with the message passing library and an awareness of the extensive optimization of MPI communications by manufacturers. Such a system would need to be multi-threaded to allow the overlap of computation and communication to mask the high-latencies, and to exploit the emergence of low-cost hardware such as multi-processor machines/multi-core processors.
Some of the main concerns have been to minimize the use of the high-latency communication channels between participating grid sites, to favour a lower message count with higher message payload, and also to avoid any need for specialised hardware support from the interconnect for remote memory access.

3. Grid DSM System Requirements

The design of a Grid S-DSM system must balance many factors, some conflicting. Some of these choices include the specification of the development interface, shared data management routines, the maintenance of shared data consistency across distributed processes, and the method of communication between DSM processes. With grid computing additional factors are introduced, e.g. grids may be composed of multiple architectures and platforms with different native data representations and formats.

3.1. Parallel Grid Applications and DSM

A DSM system must present the programmer with an easy-to-use and intuitive API so that the burden associated with the construction of a parallel application is minimized. This infers that the semantics should be as close to that of shared memory programming as possible, and that it should borrow from the successes and learn from the mistakes of previous DSM implementations.
For a DSM to gain acceptance, compliance with open standards is also necessary. Rather than having a direct implementation of one it may be more beneficial if the DSM forms the basis for a target of a parallelizing compiler that supports a programming standard, such as OpenMP. There are other projects have adopted this approach of source to source compilation [14]. Initial design requirements of the DSM can be identified by examining some OpenMP directives. The OpenMP code snippet below is an implementation the multiplication in parallel of two matrices.

```
int c[ROWSA][COLSA]
...
/*** Begin parallel section ***/
#pragma omp for
for (i = 0; i < ROWSA; i++){
  for(j = 0; j < COLSB; j++){
    c[i][j] = 0;
    for (k = 0; k < COLSA; k++)
      c[i][j] = c[i][j] +
                a[i][k] * b[k][j];
    }
  }
} /*** End parallel section ***/
```

Barrier primitives are used implicitly in OpenMP. Parallel application threads will wait at the end of the structured code block until all threads have arrived, except in some circumstances such as where a *nowait* clause has been declared. In order for concurrency to be allowed inside the parallel section, the shared memory regions must be writable by multiple writers, providing the accesses are non-competing. The code example above demonstrates the importance of allowing multiple writers to access shared data areas concurrently.
Additionally, mutual exclusion is required where parallel writes are possibly competing. In OpenMP, mutual exclusion areas are defined with the *master / single / critical* directives. A distributed mutual exclusion device (shared lock) is required to implement the *critical* directive, while the *master* and *single* OpenMP directives are functionally equivalent, and can be implemented using a simple if-then-else structure. Again implicit barriers are present at the end of these directives unless otherwise specified.

3.2. DSM Performance

The primary desire is the minimization of communication between nodes when maintaining consistency of shared data between DSM processes. Thus the selection of the consistency model is a prime determinant in the overall performance of the DSM system. This requires a relaxed consistency model to be employed, where data and synchronization operations are clearly distinguished, and data is only made consistent at synchronization points. The most notable are Release Consistency (RC) [10] Lazy-Release (LRC) [13], and Entry Consistency (EC) [6] models. The choice of which to use generally involves a trade-off between the complexity of the programming semantics, and the volume of overall control and consistency messages generated by the DSM.
Entry consistency is similar to lazy-release consistency in its use of synchronization primitives to direct coherency operations to shared data, but adopts an even more relaxed approach, whereby shared data is explicitly associated with a synchronization primitive(s) and is made consistent when such operations are performed; hence, minimal coherency messages are generated. Studies show that EC and LRC can on average produce the same performance on clusters of computers of a modest size [3]. It is important to note that the application's access pattern to shared memory is a determining factor. Entry consistency introduces additional programming complexity, but [16] examines the same application with EC and LRC, which clearly demonstrates the better performance of EC with respect to volume of messages generated. This is an important consideration if an application is executing across multiple sites with high-latency communication links.
Multiple-writer protocols attempt to increase concurrency by allowing multiple writes to locations

residing on the same coherence unit, so addressing the classical DSM problem of false sharing. Multiple-writer entry consistency attempts to combine the benefits of both models while addressing associated problems that have been identified for the LRC and EC protocols [4]. It has been demonstrated that significant performance gains can be achieved by employing this technique [18]. This protocol must be provided by the DSM in order to achieve true concurrency.

3.3. Use of a Grid Information & Monitoring System

There are tools available for the monitoring of message passing applications, such as MPICH's Jumpshot, but these are unsuitable for execution across geographically dispersed sites. It has been shown that when MDS, the grid information component of the Globus Toolkit, was integrated with a MPI implementation, dramatic improvements were obtained in the performance of a number of MPI collective operations through the use of hierarchy awareness [12]. If a Grid DSM can make use of services in this way then it should prove beneficial. Some of the benefits of integrated environmental awareness include the efficient implementation of global barrier synchronization, load-balancing using per-site lock management optimisation, effective per-site caching & write collection of shared data, and communication-efficient consistency updates.
The Relational Grid Monitoring Architecture (R-GMA) [17], is a relational grid information system, where information can be published via numerous distributed producers and accessed by a single consumer. R-GMA has successfully been used to enable the monitoring of MPI applications in a grid environment using GRM/PROVE [15]. Runtime monitoring and hierarchy awareness for DSM applications could be provided in a similar manner.

4. Implementation

Our prototype Software-DSM system is called SMG (Shared Memory for Grids). The MPI message passing library is currently utilized for system communication as it provides a stable interface, and support exists for a execution of MPI applications in a grid environment. The SMG API implementation draws on previous DSM APIs and consists of initialization & finalization function (essentially wrappers for the underlying communication routines), shared memory allocation functions, and synchronization operations. To maximise portability, the DSM implementation only uses standard libraries such as the POSIX thread, MPI, and the standard C libraries. No special compiler or kernel modifications are required for the DSM implementation itself. The development platform is the Linux operating system.
The code example below shows how the matrix multiply algorithm that was implemented in the previous section using OpenMP directives can now be implemented using the SMG library. SMG barriers appear before and after the parallel section. The resultant matrix *c* is explicitly declared and bound to the use of the synchronization variable *SMG_BARRIER_01*. When this barrier is reached the modifications to the shared variables are synchronized, checked for conflicts, and propagated among the nodes.

```
SMG_malloc(SMG_OBJ_001,sizeof(int),(COLSA * COLSB),&c_ptr,
            (ENTRY | BARRIER_NAMED), SMG_BARRIER_01);
// c initialised to c_ptr
...
SMG_barrier_(SMG_BARRIER_01);
for (i = start; i < end; i++)
  for(j = 0; j < COLSB; j++){
```

```
      c[i][j] = 0;
      for (k = 0; k < COLSA; k++)
        c[i][j] = c[i][j] +
                  a[i][k] * b[k][j];
    }
SMG_barrier_(SMG_BARRIER_01);
```

Although an OpenMP compiler targeting the SMG system has yet to be implemented it is envisaged that it would produce valid SMG code similar to the example above. Usage of SMG locks is not demonstrated here, but the single-reader/multiple-writer protocol is supported. Upon exclusive access of a lock by a process, writes can occur to the shared data that may be bound to that particular lock. Upon release of the lock and its subsequent acquire by another process all modifications made in the previous duration will be propagated to the requesting process.

4.1. Consistency and Coherency

EC increases the programming burden when compared with other relaxed consistency models such as LRC. However, its semantics can lend itself well to targeting by an OpenMP compiler, and for this reason as well as its lower message volumes it is the consistency protocol of choice. Write-update coherency is employed by default when an EC protocol is used. Multi-writer EC is supported when the shared data is associated with a barrier synchronization primitive, while lock primitives support a single-writer.

In S-DSMs, write-trapping is the process of detecting modifications made to shared data, while write-collection is the process by which the updates required to maintain consistency at remote processes are generated. The implementation of these concepts follows a similar approach to that adopted in Treadmarks [13] with some small modifications. Write-trapping is achieved by setting the protection level of the shared object, where the minimum granularity is at the virtual page level up to the total size of the object if it occupies more than one page. Upon the first write to a variable in a shared region a twin, or part thereof, is generated.

When the synchronization object that the shared memory object is bound to performs a release a diff is generated, by comparing the twin and the current 'dirty' state. This is used to minimize the message traffic for coherence updates. If topology information is available then hierarchy awareness can be applied in barrier operations, i.e. as arrival notifications are received the diffs can be merged at intermediate nodes, thus reducing the processing bottleneck at the root node level. Otherwise, a traditional tree-structured barrier is employed.

Multi-writer entry consistency is supported for barrier primitives as demonstrated in the code segment above; this is equivalent to OpenMP parallel *for* constructs. Basic user-specified alteration to write trapping and write collection methods are allowed, and when more integration with the information system occurs, this user control will enable application-level optimizations where access patterns to shared data are irregular [4].

4.2. Communication

Communication between processes is via an implementation of the SMG_comm interface. Thus far only a MPI version of this has been developed. Exploring the use of MPICH for the communication between DSM processes executing on distributed nodes allows for the exploitation of an optimized and stable message passing library, and also leveraging of useful MPI resources such as profiling tools and debuggers; its use also insulates the system from some platform dependencies and will ease porting to other architectures and platforms in the future.

Unfortunately the current Grid enabled version of MPICH, MPICH-G2, is based on the MPICH distribution (currently version 1.2.7), which has no support for multi-threaded MPI applications. This makes optimisations such as hybridizing (see 4.4) near impossible, as the DSM system thread requires the MPI communication channel, and so can only be used in MPI_THREAD_FUNNELLED mode. Implementations exist that provide a thread-safe MPI implementation, however they are not grid-enabled. Other MPI implementations are soon expected to support multi-threading.

4.3. Environment Awareness

An information system is required if the DSM is to be hierarchy-aware, otherwise the topology assumes the flat model typical of an MPI application. Both MDS and R-GMA adhere to the GLUE schema [2]. Either (or both) are a good basis for hierarchy awareness. The SMG DSM prototype initially uses R-GMA, principally for its support for relational queries and dynamic schemas. It uses the GLUE schema to obtain environmental information from GLUE compliant R-GMA information producers. To enable monitoring of the user applications, SMG schemas are defined so that monitoring information can be published using the R-GMA producer API, and viewed using the standard R-GMA browser. Nonetheless the information system is hidden by wrappers, and other systems, such as MDS, can be used by implementing the required wrapper APIs.
The SMG system makes use of R-GMA to produce and consume data in a similar manner. Topology information is consumed at startup, while DSM system and application information can be produced using the defined APIs. A tool has been developed to access application logging information allowing for runtime analysis. A further aim has been to use the information system to create topology/hierarchy awareness and runtime support of applications, where the developer wishes topology information to be exposed to the user application code to allow for optimizations at that level.

4.4. Incremental Hybridization

Using the information system allows for the runtime profiling of user applications, and allows for other tools such as deadlock detectors to be developed. We have constructed a user interface tool that can identify locations in an application where data access behaviour results in severe performance penalties. As we can monitor the variables and the code areas where these are used we can direct a *hybridization* process whereby the user replaces shared memory code with message passing, resulting in performance gains. This can be done in a localized fashion, and possibly in the future in conjunction with parallelization with OpenMP.
The hybridization GUI works by directing the user to the locations where the majority of communication occurs (so identifying the participating processes), and also to the shared variables that are responsible for the communication. The code browser illustrates the 'hot-spots' in the application and the the objects responsible. This allows for incremental and localized optimisations.
The developer can examine the interval between synchronization operations, such as between barriers, and view the volume of data transfers generated between processes. From other information gathered during the interval they can identify the memory objects causing the communication.If message passing and shared memory paradigms are used at different times during the application's execution for the same variables, then the shared regions must be made consistent after message passing is used.
Another feature is the highlighting of fragmented shared memory use which may be the result of poor algorithm design/implementation. This can be identified when a shared memory region is repeatedly modified by multiple processes in a fragmented fashion. Such a scenario may occur when a developer, unaware of the potential differences between C and Fortran languages (row and column ordering) transposes a matrix multiplication algorithm.

5. Performance

There is no grid-enabled MPI implementation that currently supports multi-threaded applications, we therefore utilize a non-grid flavour of MPI. The SMG DSM is currently being tested in a virtual grid environment, with the number of sites configurable at runtime. System performance measurements are not representative, since DSM communication is unable to avail of blocking receive MPI calls. However, the significant memory overhead associated with the DSM (update catalogue) persists, as will the substantial processing & memory requirements for consistency actions (twinning/diffing). This can be overcome at the expense of increased coherence message volumes.
Table 1 below shows the total message count resulting from the SMG implementation of a simple Laplace example compared with that of a similar message passing implementation, each involving the same number of iterations. The message volumes compare favourably with the difference being extra messaging incurred by the DSM at initialisation & finalization. Additionally, the usefulness of some of the strategies we employ (such as hierarchy-awareness) can be demonstrated; where the simulation involves *N* sites, the number of inter-site messages per barrier is maintained at *2N*.

Number Processes	4	8	16	32
MPI	603	1407	3015	6231
SMG	612	1428	3060	6343

Table 1
Messaging costs for parallel Laplace

6. Conclusions and Summary

Grids are starting to impact on mainstream parallel computing. If this trend is set to continue then improved tools and development environments must be implemented. We believe that there will be numerous approaches to constructing grid applications, be it message passing, shared memory, or DSM. Clearly none of these in isolation will provide the perfect fit, but rather an ensemble.
DSM implementations such as Treadmarks, Munin, and Midway were written for compute clusters, not for Grid computing. Efforts at making hierarchy-aware DSM consistency protocols do exist [5], as well as efforts to allow OpenMP to run on distributed memory machines [14]. There have been other attempts at providing a shared memory model for wide area computing [8], and efforts are underway to implement the MPI-2 standard, which includes specifications for remote memory access and one sided communications.
In the SMG system, we are attempting to demonstrate the potential advantages when message-passing and DSM programming paradigms are combined in the grid environment. The goal is to reduce the programming burden, and allow it to be followed by incremental optimisation. If this is achieved, it will promote the use of grids by allowing the exploitation of the very large collection of existing shared memory codes, and allow for easier parallelization/grid-enabling of serial codes through the use of the OpenMP standard. Coherence overheads are comparable with other DSM implementations but this will be improved when environmental awareness techniques and alterable write trapping/collection techniques are further improved.
Future support for heterogeneity is vital in order to further the cause. This problem is non-trivial, as shared data would need to be strongly typed. We believe that this is one area where compiler support would prove beneficial.

References

[1] Homepage of OpenMP initiative. http://www.openmp.org.

[2] The GLUE Schema. http://www.cnaf.infn.it/sergio/datatag/glue/index.htm.

[3] S. Adve, A.L. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel. A Comparison of Entry Consistency and Lazy Release Consistency Implementations. In *Proceedings of the Second High Performance Computer Architecture Conference*, pages 26–37, Feb 1996.

[4] Christina Amza, A.L. Cox, Sandhya Dwarkadas, Li-Jie Jin, Karthick Rajamani, and Willy Zwaenepoel. Adaptive Protocols for Software Distributed Shared Memory. *Journal of the IEEE, Special Issue on Distributed Shared Memory*, pages 467–475, Mar 1999.

[5] Gabriel Antoniu, Luc Boug, and Sbastien Lacour. Making a DSM Consistency Protocol Hierarchy-Aware: an Efficient Synchronization Scheme. In *Proc. Workshop on Distributed Shared Memory on Clusters (DSM'03)*, pages 516–523, May 2003.

[6] B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon. The midway distributed shared memory system. In *Procs. of COMPCON. The 38th Annual IEEE Computer Society International Computer Conference*, pages 528–537, Feb 1993.

[7] J. B. Carter, D. Khandekar, and L. Kamb. Distributed Shared Memory: Where We Are and Where We Should Be Headed? In *Fifth Workshop on Hot Topics in Operating Systems*, pages 119–122, 1995.

[8] DeQing Chen, Chunqiang Tang, Xiangchuan Chen, Sandhya Dwarkadas, and Michael L. Scott. Multi-level Shared State for Distributed Systems. In *Procs. of 31st Int. Conference on Parallel Processing (ICPP'02)*, August 2002.

[9] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan-Kaufmann, 1999.

[10] K. Gharachorloo, D. Lenoski, J. Laudon, P.B. Gibbons, A. Gupta, and J.L. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *25 Years ISCA: Retrospectives and Reprints*, pages 376–387, 1998.

[11] N. Karonis, B. Toohen, and I. Foster. MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing*, 63(5):551–563, May 2003.

[12] N. T. Karonis, B. R. de Supinski, I. Foster, W. Gropp, E. Lusk, and J. Bresnahan. Exploiting Hierarchy in Parallel Computer Networks to Optimize Collective Operation Performance. In *Proc. of the 14th Int'l Conference on Parallel and Distributed Processing Symposium (IPDPS-00)*, pages 377–386, 2000.

[13] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proc. of the Winter 1994 USENIX Conference*, pages 115–131, 1994.

[14] H. Lu, Y. C. Hu, and W. Zwaenepoel. OpenMP on network of workstations. In *Proc. of Supercomputing'98*, 1998.

[15] N. Podhorszki and P. Kacsuk. Monitoring Message Passing Applications in the Grid with GRM and R-GMA. *Recent Advances in Parallel Virtual Machine and Message Passing Interface (PVM/MPI)*, pages 603–610, 2003.

[16] Jelica Protic and Veljko Milutinovic. Entry consistency versus lazy release consistency in dsm systems: Analytical comparison and a new hybrid solution. In *IEEE Workshop on Future Trends of Distributed Computing Systems*, pages 78–83, Oct 1997.

[17] S. Fisher et al. R-GMA: A Relational Grid Information and Monitoring System. Technical Report WP3-2003-01-14, 2nd Cracow Grid Workshop, DATAGRID, Jan 2003.

[18] H. S. Sandhu, T. Brecht, and D. Moscoso. Multiple Writers Entry Consistency. In *Proc. of the Int'l Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, volume I, pages 355–362, 1998.

[19] E. Speight, H. Abdel-Shafi, and J.K. Bennett. An Integrated Shared-Memory/Message Passing API for Cluster-Based Multicomputing. In *Procs. of the Second International Conference on Parallel and Distributed Computing and Networks (PDCN)*, Brisbane, Australia, 1998.

Hierarchical and Reliable Multicast Communication for Grid Systems

Nadia Ranaldo[a], Giancarlo Tretola[a], Eugenio Zimeo[a]

[a]RCOST - University of Sannio - 82100 Benevento, Italy

Grid computing is emerging in recent years as a viable computing paradigm to solve data and compute-intensive distributed applications. Most of these applications need to transfer the same large amount of data to a wide collection of resources, typically shared among multiple organizations. This paper analyzes the performances of a reliable multicast protocol and the impacts derived from its integration in a Grid middleware platform to efficiently implement one-to-many communication mechanisms easily usable at programming level. At this level, object-oriented, typed groups are used by programmers to transparently exploit the features of the underlying Grid middleware. This way, if the middleware takes into account a hierarchical organization of resources, scalable master-slave applications can be easily written and efficiently executed by using hierarchical groups.

1. Introduction

Thanks to the huge amount of resources available across the Internet and to improvements of wide-area network performance, computational, data and service Grids are becoming effective distributed infrastructures to execute high-performance, general purpose, data and compute-intensive applications. Most of these applications (such as simulations applied to scientific and engineering fields, or data acquisition and analysis from distributed measurement instrumentation and sensors, etc.) deal not only with intensive computations, but also with the management of huge amounts of data that often are transferred to a wide collection of resources. The majority of existing middleware platforms (Globus [9], UNICORE [10], etc.) typically adopt unicast communication mechanisms implemented, independently of the underlying physical network, atop unicast reliable protocols (such as TCP) to perform data transmission. We think that better performance could be achieved through more efficient communication mechanisms. The idea is to perform the data transmission leveraging optimized services of underlying Grid middlewares to exploit the potentialities of the network connections actually available. For instance, a transport layer based on IP multicast should be exploited as an alternative to the commonly used unicast transport communication based on TCP to reduce network traffic and communication overheads in some cases: resources connected through bus-based LANs that support broadcast communication at data-link layer, or a set of workstations directly connected to an IP multicast-enabled router, or indirectly connected through the tunnelling technique.

Using reliable multicast protocols in Grid computing is still an open issue and a lot of researches have been made in recent years [15] [2]. Most of these have aimed at easily porting existing Grid applications to multi-destination environments by enriching TCP with multicast capabilities (*protocol level*) [11] [13] [16]. However, it is worth noting that typically Grid middleware platforms exhibit their own programming interfaces that hide low-level communication APIs, such as sockets. Therefore, TCP extension is useful when an existing application that directly uses TCP has to be modified in order to deliver data to multiple receivers. Another approach could be based on application-aware components, called Active Routers [14], disseminated in specific points of the Grid infrastructure, which are able to handle application-dependent services on incoming data packets (*infrastructure level*), for example to improve the performances of a multicast communication. This approach has

the disadvantage of requiring the deployment of specific routers, with ad-hoc execution environments, that limits its application and widespread in the implementation of real Grid systems.

In this paper, we propose an approach based on the integration of a reliable multicast protocol into a Grid middleware platform (*middleware level*). The paper discusses the impacts of multicast on the applications by using HiMM (*Hierarchical Metacomputer Middleware*) [7], a Java-based middleware for Grid computing, which delivers basic services of computation, information, communication and resource management to write and execute parallel and distributed object-oriented applications leveraging a virtual hierarchical computing environment. However, the problems and solutions proposed can be easily applied to other Grid middleware platforms.

At programming level, the paper focuses on the requirements for easily programming and executing applications based on the master-slave model, known also as task-farming, or master-worker model. Such model is particularly suited to program in heterogeneous Grid environments [3] where the main issues are: (1) easy-to-use and high-level application programming interface and (2) high performances through an efficient implementation on the available computing and network infrastructures.

To tackle the above issues, we think that group communication mechanisms can be adopted to easily write applications according to the hierarchical master-slave model. As a consequence, providing a middleware for Grid computing with an effective and efficient implementation of the group abstraction could simplify software development and reduce the communication overhead both in small scale and in large scale networks.

The rest of the paper is organized as follows. In Section 2, the integration of a reliable multicast protocol in HiMM is described. Section 3 presents the extended ProActive groups to easily write master-slave applications, and in Section 4 some experimental results are reported. Finally, Section 5 concludes the paper and introduces future work.

2. Reliable Multicast for Hierarchical Grid Computing

HiMM implements basic asynchronous point-to-point and point-to-multipoint communication mechanisms which exploit the hierarchical organization of a Grid hardware architecture. HiMM allows an application to asynchronously send a message to (1) a node at the same level of the sender (*send*); (2) all the nodes at the same level of the sender (*limited broadcast*); (3) all the nodes at the same level of the sender and all the other nodes recursively met in each macro-node belonging to the level of the sender (*deep broadcast*).

All these primitives are implemented on top of a transport layer which provides a one-way communication interface for sending objects (in unicast and broadcast mode). This layer permits to transparently use different transport protocols or other connectivity services, on the basis of the availability. As a consequence, a reliable multicast protocol can be integrated in the multi-protocol transport layer of HiMM to efficiently implement the point-to-multipoint communication mechanisms provided by the primitives “limited broadcast” and “deep broadcast”. When a broadcast primitive is invoked on a node, many partial broadcast invocations take place, one for each protocol domain (PD). A PD is the set of nodes that use the same protocol to communicate.

Each transport adapter provides a specific implementation of the broadcast primitives, which performs point-to-multipoint communication as fast as possible for the specific PD. Each PD can manage one of two possible communication approaches, unicast or multicast. With the first approach, the protocol adapter transmits the data to all nodes of a PD using a sequence of unicast messages; with the second approach, the protocol adapter exploits a reliable multicast protocol to send the same data to the nodes of a PD using a single multicast message.

We implemented and integrated an additional protocol adapter for a reliable multicast protocol that exploits native multicast communication over IP. Among the reliable multicast protocols proposed in the literature, we chose to use TRAM (*Tree-based Reliable Multicast*) [5] because of its tree-based organization of nodes that represents the most scalable method for supporting multicast communication on large scale Grid systems. In fact, TRAM was designed to support the transfer of a large quantity of data and to manage a large number of receivers without affecting the sender activity [8]. The tree-based strategy is the most efficient one for implementing reliable multicast protocols for Grid applications based on the hierarchical master-slave pattern.

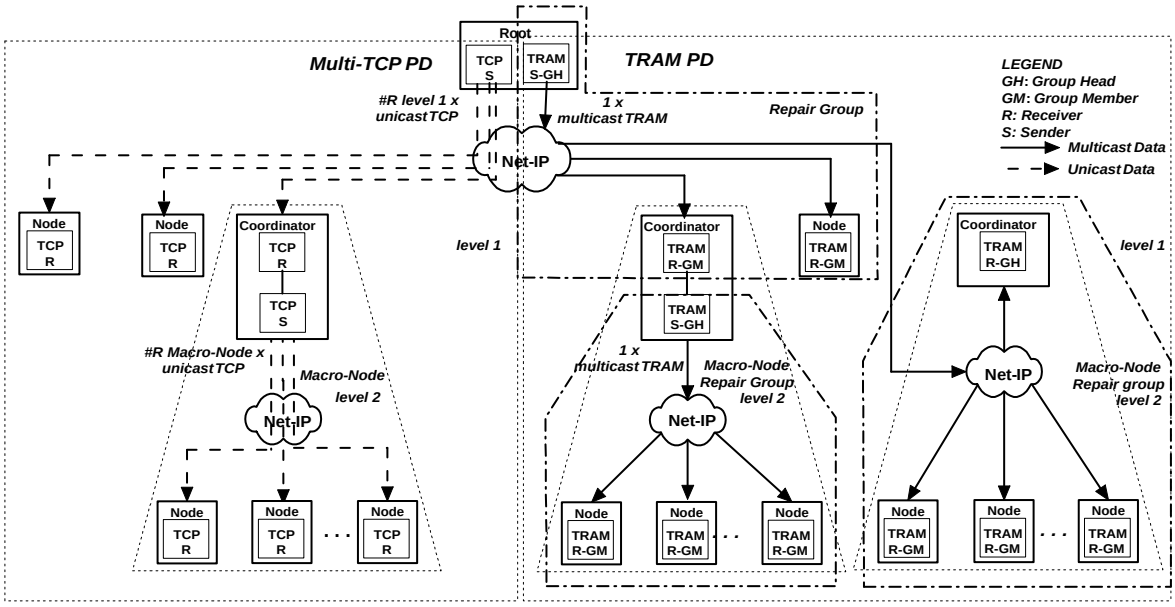


Figure 1. Deep broadcast over a hierarchical architecture with two PDs

According to the hierarchical schema, TRAM ensures reliability by means of a local error recovery mechanism based on repair groups organized according to a tree structure. Each recovery group has a receiver that represents the group head that has to store every message received from the sender in order to allow message recovery when some members of its group notify losses. All the group members receive multicast data from the sender and periodically send ACK messages to their group heads for notifying messages reception or losses. TRAM builds the tree so that group heads are close to their group members. The tree formation is dynamic and is based on TTL values which permits to minimize network bandwidth consumption. On the other hand, this mechanism has some limitations since current multicast routing protocols do not provide accurate TTL values. Moreover, not every group member may be suitable to perform the role of group heads, due to hardware, software or administrative limitations. In HiMM these drawbacks can be overcome if the TRAM repair tree is mapped on a hierarchical structure in which each level includes a special node, the *coordinator*, dedicated to special management functionalities.

Figure 1 shows an example of deep broadcast primitive invoked by the root node of a hierarchical network to send the same data to all the nodes of the system, for example for a master-slave computation. The figure shows two PDs, the Multi-TCP PD and the TRAM PD. The Multi-TCP PD uses the

TCP transport adapter to typically manage communication primitives on the HiMM nodes installed on machines which do not support native IP multicasting. The TRAM PD uses the TRAM transport adapter to efficiently implement the broadcast primitives. Thanks to the mechanisms supported by TRAM for the repair tree formation, each HiMM coordinator can be configured as group head for the nodes of the managed macro-node, while the nodes can be simple group members. The root is also a group head because it is the sender node. On the basis of the effective physical connections among the nodes, the TRAM transport adapter can manage two different cases: (1) a coordinator used as front-end of a pool of machines not directly accessible by the user machine; (2) a coordinator connected to the managed nodes through an IP-multicast enabled network. In the first case, the deep broadcast mechanism requires the coordinator to create its own repair tree in order to reach the nodes of the macro-node. In the second case, the nodes of the macro-node can receive data directly from the sender, and the coordinator, configured as a group head, can be used from the nodes to recover lost data, so reducing the work-load on the sender.

3. Master-slave Computing over Reliable Multicast

The original master-slave pattern [4] may cause scalability problems when it is implemented in a geographically distributed environment and does not give the master the direct visibility and access to the slaves hidden behind front-end computers. A hierarchical version of the master-slave pattern better fits in the architectural features of large Grid systems. In this case, the master at the top controls the overall computation and distributes it among the masters at lower levels, and so on, until the computation is sent to slaves that directly process the request. The collection of results is performed in the reverse order.

To support the master-slave computational model and its hierarchical version, in [1] we proposed to use groups at programming level through an extension of object oriented, typed groups provided by ProActive [6] that allows for the programmability of group behaviors. An extended ProActive group is a group whose behavior can be dynamically configured by means of a set of group semantics, which specify both the behavior of the group and the logical communication models to apply during a method invocation (unicast, multicast and broadcast, that can be unreliable or reliable). Such semantics are (1) *request mapping*, which determines the policy of request dispatching to the group members; (2) *input distribution*, used to determine how to manage the actual input parameters of a method invocation; (3) *synchronization*, used to define the synchronization policy when a method invocation has to return a result; (4) *output collection*, which determines how to reply to the caller the final return value of a group method invocation. The extended group mechanism can be adopted to easily implement the hierarchical master-slave pattern as a generic template to directly use for programming applications. To this end, the four semantics are to be specified as follows: (1) a request has to be dispatched to all the slaves to leverage the overall computational power of the distributed resources; (2) the workload is divided among the slaves by using a scatter policy; (3) the policy has to be “a wait for all the results” since the final result will be the aggregation of all the partial results coming from the members; (4) the final result is built assembling the partial results of the slaves. For the constructor, the input distribution semantic can be implemented with the broadcast policy, since all the members must be the same objects.

4. Experimental Results

To implement multicast communication in HiMM we used the Java-based JRMS framework (*Java Reliable Multicast Service*) [12], which currently provides reliable multicast based on LRMP and

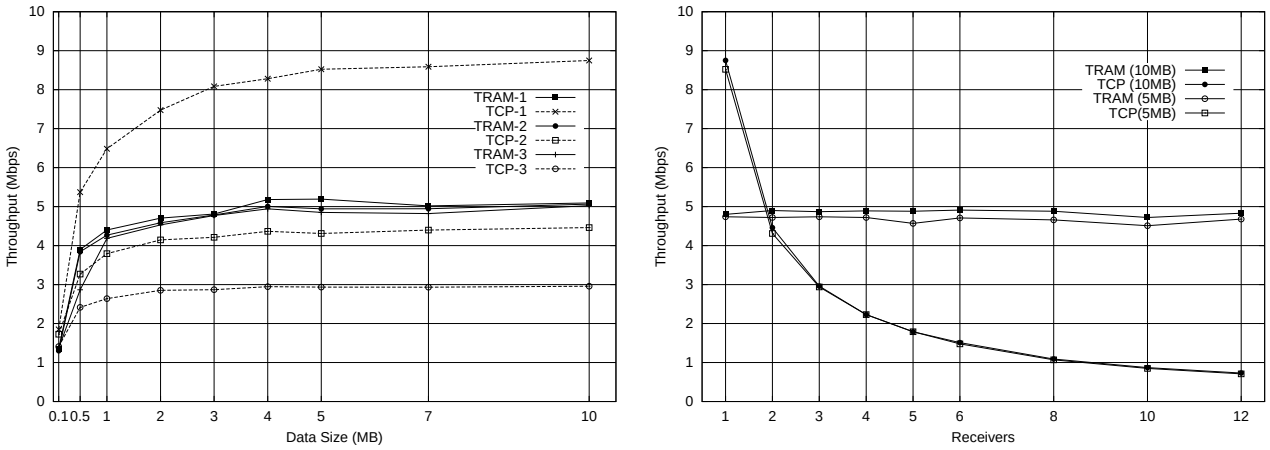


Figure 2. (a) TCP and TRAM throughputs; (b) TRAM scalability

TRAM protocols, but can be easily extended to support other protocols. In this section we analyze the throughput and scalability of TRAM and we show the performance comparison of a master-slave application developed adopting, for the group member creation, the default ProActive group mechanism, based on RMI (which in turn uses the TCP unicast communication mechanism to communicate with each group member), and the extended ProActive group mechanism based on TRAM.

4.1. TRAM throughput and scalability analysis

A first experiment was conducted considering a cluster of eight nodes, each one equipped with two Intel Pentium II 350 MHz, 128 MB RAM and 10/100 Mbps network card, running Microsoft Windows 2000 operating system. The nodes were interconnected through a hub 3COM TP16C at 10 Mbps, and the maximum data rate for the TRAM flow control protocol was fixed to 1.25 MBps, a value that was proven [8] to deliver the best performance. Figure 2.a shows the actual throughput obtained by TRAM and TCP to transfer the same amount of data to one, two and three receivers. This throughput is measured as the data amount divided by the total time to transfer data to all the receivers without losses, where the total time is calculated as the arithmetic mean of multiple measurements. In the case of TCP, the total time is evaluated considering a unicast transmission for each receiver. TRAM delivers a throughput of about 5 Mbps, which is not significantly affected by data dimension and by the number of receivers. TCP delivers a lower throughput starting from two receivers. Figure 2.b shows the throughput obtained by TRAM and TCP to transfer 10 MB and 5 MB of data to a varying number of receivers (from one to twelve). The figure shows a nearly constant trend of the throughput of TRAM with respect to an increasing number of receivers, which proves the good performance of TRAM and its useful adoption as an alternative to TCP for reliable communication mechanisms in Grid computing. Moreover, it is worth noting that the performances of TRAM are not significantly influenced by the message dimension. We stress the fact that the bandwidth efficiency of TRAM is about 50% and it is nearly independent of the number of receivers.

A second experiment was conducted considering the same cluster of nodes interconnected through a switch HP Procurve 2524 at 100 Mbps to evaluate the total transfer times, throughput and scalability of TRAM. In this case the maximum data rate was fixed to 12.5 MBps. Because of the high available physical bandwidth, we pointed up re-transmissions, which instead we did not notice in the previous experiment. Such re-transmissions affect the average total transfer times and as a conse-

quence the throughput. Figure 3.a shows the throughput obtained by TRAM and TCP to transfer the same amount of data to a varying number of receivers: the dotted curve represents the throughput of TRAM obtained in absence of re-transmissions. This curve demonstrates, as for the previous case, a nearly constant trend of the TRAM throughput, which is about 20 Mbps, therefore the bandwidth efficiency is only 20% of the maximum bandwidth. Moreover the figure indicates that in this case TRAM gives better performance than TCP only from five receivers on.

Finally, in a third experiment a network of computers, interconnected through the same switch at 100 Mbps, was used to evaluate the TRAM performance considering heterogeneous and more powerful computers in a non-dedicated environment with a time-varying bandwidth availability. A notebook Pentium M 1.6 GHz with 512 MB RAM was used as sender, and six computers were used as receivers, in particular: a Pentium IV 1.8 GHz with 256 MB RAM, two Pentium IV 2.0 GHz with 512 MB RAM, a Pentium IV 2.4 GHz with 512 MB RAM, two AMD Athlon XP 1800+ 1.5 GHz with 384 MB RAM. All the computers ran Microsoft Windows XP operating system.

Figure 3.b shows the actual throughput obtained by TRAM and TCP to transfer 10 MB of data to a varying number of receivers. It is worth noting that, while TCP is not significantly affected by hardware features, using computers with more CPU speed and RAM capability, TRAM achieves a higher throughput (about 24 Mbps). Also in this case we pointed up re-transmissions in TRAM and, as for the previous case, a nearly constant trend of the TRAM throughput with respect to the number of receivers.

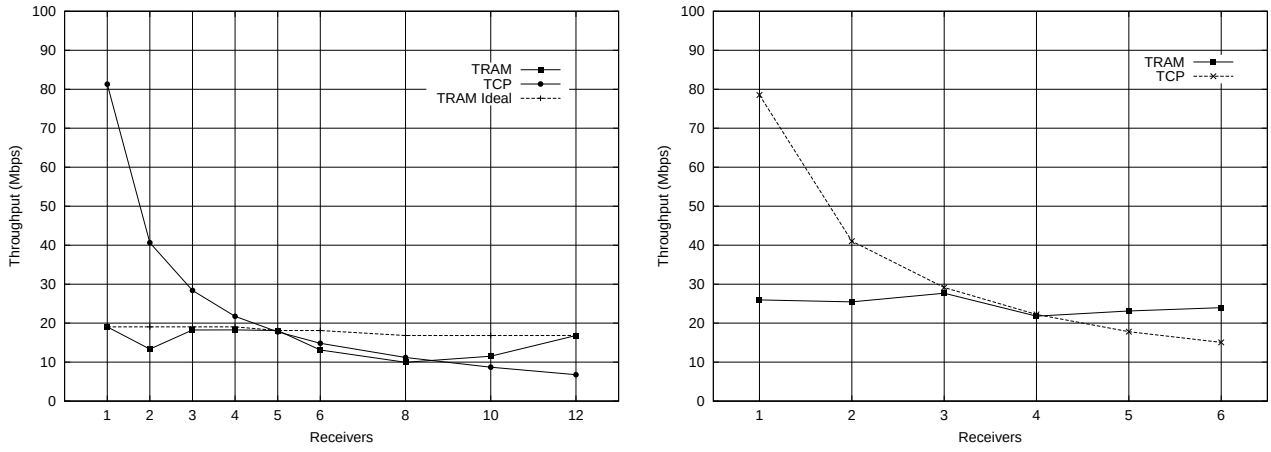


Figure 3. TRAM scalability on a 100 Mbps network of (a) slow homogeneous computers; (b) fast heterogeneous computers

Considering the previous experiment results, we can state that TRAM-based multicast communication can be adopted instead of TCP-based unicast communication over the Internet in order to increase the throughput, whereas for dedicated high-performance networks, the advantage in using TRAM depends on the number of receivers and on the features of adopted network and machines. We are currently studying the parameters and experimental conditions which could affect re-transmissions and the TRAM data flow control in order to improve the TRAM performance and eventually individuate improvements and extensions for specific physical networks and machines.

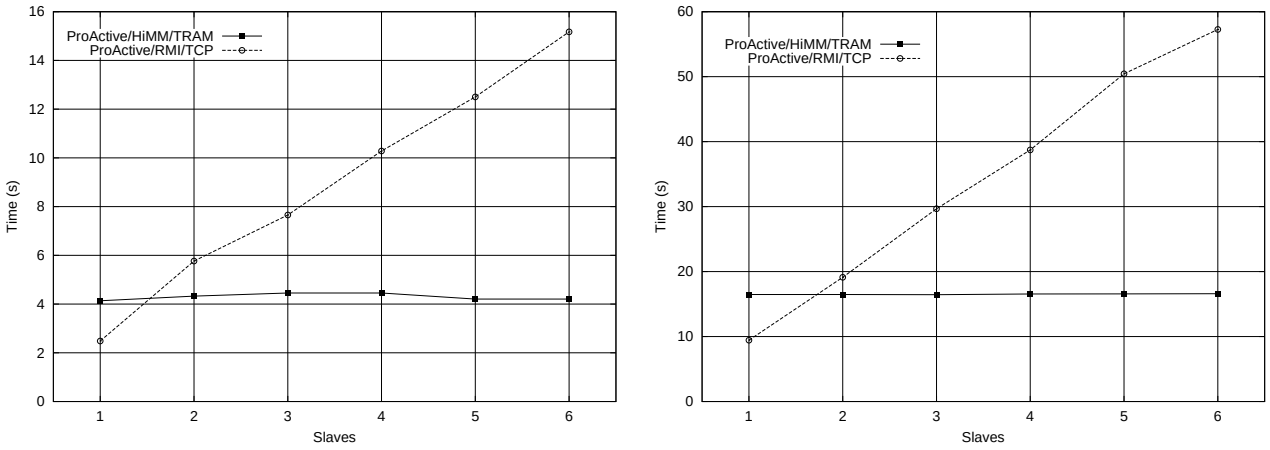


Figure 4. Transfer times for the right matrix of (a) 800x800; (b) 1600x1600

4.2. Master-slave application performance analysis

An experimental analysis of a master-slave application (parallel matrix multiplication) implemented both with extended and native ProActive groups was performed on the first testbed described above and characterized by a 10 Mbps network.

Figure 4 shows the transfer times for the creation of group members, which include the transmission of the right matrix to a varying number of slaves. The native ProActive groups use repeated unicast TCP transmissions whereas the extended Proactive groups leverage the TRAM transport adapter, and so the time for group creation is almost independent of the number of receivers. Figure

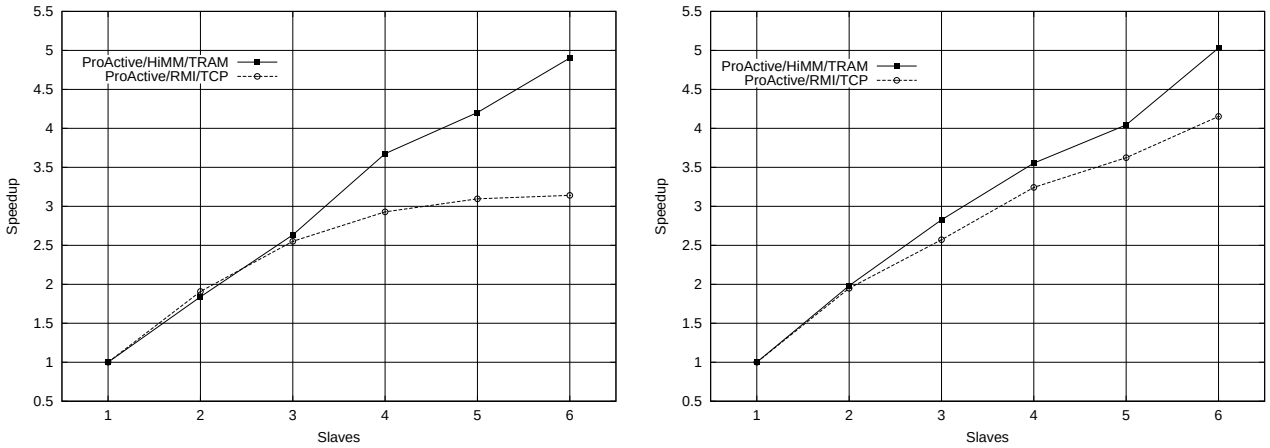


Figure 5. Speedup evaluation for matrices of (a) 800x800; (b) 1600x1600

5 shows relative speedups measured considering the total execution time for the matrix multiplication. The implementation based on reliable multicast exhibits better performances, mainly due to the reduced traffic on the network during the group creation phase. The distribution of the left matrix

and the results collection are performed through unicast communication.

5. Conclusions and Future Work

The paper presented and discussed the impacts derived from using a reliable multicast protocol for the communication in a Grid environment when at application level the master-slave model is adopted. Such model was implemented with the group communication mechanisms provided by an extension of Proactive groups whose performance improvements were clearly demonstrated by the experimental results. An analysis in larger distributed environments will be performed as future work to evaluate the advantage of the hierarchical organization of TRAM on large scale systems. An additional improvement of TRAM will be possible taking into account the structure of the repair-tree, its formation algorithms and other parameters related to the management of the repair buffers.

References

[1] L. Baduel, F. Baude, N. Ranaldo, E. Zimeo. Effective and Efficient Communication in Grid Computing with an Extension of ProActive Groups. Proceedings of 19th IEEE International Parallel and Distributed Processing Symposium, Colorado, 2005.

[2] M. P. Barcellos, M. Nekovee, M. Daw, J. Brooke, S. Olafsson. Reliable Multicast for the Grid: a Comparison of Protocol Implementations. Proceedings of the UK E-Science All Hands Meeting, Nottingham (UK), 2004.

[3] F. Berman. High-Performance Schedulers. in I. Foster and C. Kesselman, editors, The Grid: Blueprint for a New Computing Infrastructure, Chapter 12, Morgan Kaufmann Publishers, July, 1998.

[4] F. Bushmann et al. Pattern-Oriented Software Architecture: A System of Patterns. J. Wiley and Sons, 1996.

[5] D.M. Chiu, S. Hurst, M. Kadansky, J. Wesley. TRAM: A Tree-based Reliable Multicast Protocol. Sun Microsystems Laboratories. SMLI TR-98-68, 1998.

[6] D. Caromel, W. Klauser, J. Vayssiere. Towards Seamless Computing and Metacomputing in Java. Concurrency: Pract Exp., 10(11-13), pp.1043-1061, 1998.

[7] M. Di Santo, N. Ranaldo, E. Zimeo. A Broker Architecture for Object-Oriented Master/Slave Computing in a Hierarchical Grid System. Parallel Computing , Elsevier, Dresda, Germany, September, 2003.

[8] G. Fortino, W. Russo, E. Zimeo. Reliable Multicast Protocols for Java-based Grid Middleware Platforms. IASTED Parallel and Distributed Computing and Systems, California, USA, November, 2003.

[9] I. Foster, C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. International Journal of Supercomputer Applications, Sage Publications, 11(2) pp. 115-128, USA, 1997.

[10] V. Huber. UNICORE: A Grid Computing Environment for Distributed and Parallel Computing. Proceedings of 6th Int. Conf. on Parallel Computing Technologies, Springer, pp. 258-266. Russia, 2001.

[11] K. Jeacle, J. Crowcroft. Reliable High-speed Grid Data Delivery using IP Multicast. Proceedings of UK E-Science All Hands Meeting, UK, September, 2003.

[12] S. Hanna, M. Kadansky, P. Rosenzweig. Java Reliable Multicast Service Overview. Sun Microsystems Laboratories, SMLI TR-98-68, September, 1998.

[13] S. Liang, D. Cheriton. TCP-SMO: Extending TCP to Support Medium-Scale Multicast Applications, Proceedings of IEEE INFOCOM, pp. 1356-1365, 2002.

[14] G. Rajappan, M. Dalal. Reliable Multicast with Active Filtering for Distributed Simulations. Proceedings of Military Communications Conference (MILCOM 2003), Boston, MA, October, 2003.

[15] B. Van Assche. IP Multicast for PVM on Bus Based Networks. Proceedings of Parallel Computing, Elsevier, pp. 403-410, 1998.

[16] V. Visoottiviseth et al. M/TCP: The Multicast Extension to Transmission Control Protocol. Proceedings of ICACT 2001, Muju, Korea, February, 2001.

Building Interoperable Grid-aware ASSIST Applications via Web Services*

M. Aldinucci[a], M. Danelutto[b], A. Paternesi[b], R. Ravazzolo[b], M. Vanneschi[b]

[a]Inst. of Information Science and Technologies (ISTI) – CNR, Via Moruzzi 1, I-56124 Pisa, Italy

[b]Department of Computer Science, University of Pisa, Largo B. Pontecorvo 3, I-56127 Pisa, Italy

Abstract: The ASSIST environment provides a high-level programming toolkit for the grid. ASSIST applications are described by means of a coordination language, which can express arbitrary graphs of modules. These modules (or a graph of them) may be enclosed in components specifically designed for the grid (GRID.it components). In this paper we describe how ASSIST modules can be wired through standard Web Services, and how GRID.it components may be made available as standard Web Services.
Keywords: Grid, Web Services, components, ASSIST, GRID.it

1. Introduction

The idea of "software as a service" has recently gained more and more importance because of standardization of the way in which software may be delivered as a service over the network. Once software applications are available as a service, a composite service can be created by somehow connecting services one another under a centralized or distributed control. As an example, a single client may access a set of services programmatically using some scripting language realizing a composite service. This composite service expresses a business process capturing a particular intra or inter enterprise workflow. BPEL (Business Process Execution Language) is one popular scripting language enabling the composition of services [12].

Grid technologies have also been aligned with Web services technologies to capitalize on desirable Web services properties, such as service description and discovery, automatic generation of client and server code from service descriptions, binding of service descriptions to interoperable network protocols, compatibility with emerging higher-level open standards, services and tools, as well as broad commercial support. The Open Grid Services Architecture (OGSA) is a paradigmatic example [10]: the recent version of OGSA/Globus implements the emerging standard Web Services Resource Framework (WSRF) as specified by OASIS [7]. It allows interconnecting state full resources via Web services and includes APIs for the management of transient application life cycles and event notifications. Indeed, using Globus directly is difficult because the process requires manually writing and arranging multiple XML-configuration files. These files must cover explicit declaration of all resources, the services used to connect to them, their interfaces and the corresponding bindings to the employed protocol, since Globus applications should be accessible in a platform-and language-independent manner [9].

In addition, in order to build efficient grid-aware applications, programmers must design highly concurrent algorithms that can execute on large-scale platforms. They must then implement these algorithms correctly and efficiently. Therefore, we envision a layered, high-level programming model for the grid. In such software architecture, the bottom tiers should cope with key grid requirements for protocols and services (connectivity protocols concerned with communication and authentication, resource protocols concerned with negotiating access to individual resources) and collective

*This work has been supported by the Italian MIUR FIRB *GRID.it* project No. RBNE01KNFP.

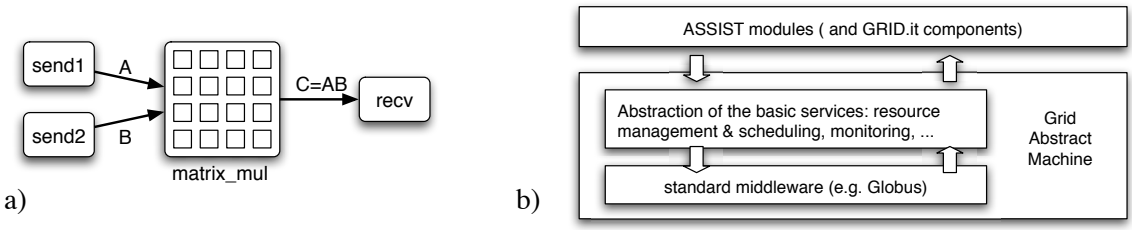


Figure 1. a) A simple ASSIST application (parallel matrix multiplication). b) ASSIST architecture.

protocols and services (concerned with the coordinated use of multiple resources) [11]. Concerning the top tiers, we envision a grid-aware application as the composition of a number of coarse grained, cooperating components within a high-level programming model, which is characterized by a high-level view of compositionality, interoperability, reuse, performance and application adaptivity. Applications are expressed entirely on top of this level. This vision is currently pursued by several research initiatives and programming environments, among the others, within the ASSIST (GRID.it project) [18] and GrADS [16] projects. The underlying idea of these programming environments is moving most of the grid specific efforts needed while developing high-performance grid applications from programmers to grid tools and run-time systems. This leaves programmers the responsibility of organizing the application specific code and the programming tools (i.e. the compiling tools and/or the run-time system) the responsibility of properly interacting with the grid.

In this work, we discuss two distinct but related kind of tools: those making available a whole ASSIST program or an ASSIST parallel module as a standard Web Service, and those supporting access to standard Web Services from within standard ASSIST code. Overall, these tools guarantee interoperability between the ASSIST and the Web Service worlds.

2. The ASSIST coordination language

ASSIST applications are described by means of a coordination language, which can express arbitrary graphs of modules, interconnected by typed streams of data. Each stream realizes a one-way asynchronous channel between two sets of endpoint modules: sources and sinks. Data items injected from sources are broadcast to all sinks. All data items injected into a stream should match stream type. A simple application implementing parallel matrix multiplication is shown in Fig. 1 a).

Modules can be either sequential or parallel. A sequential module wraps a sequential function. A parallel module *(parmod)* can be used to describe the parallel execution of a number of sequential functions that are activated and run as *Virtual Processes* (VPs) on items arriving from input streams. The VPs may synchronize with the others through barriers. The sequential functions can be programmed by using a standard sequential language (C, C++, Fortran).

A *parmod* may behave in a data-parallel (e.g. SPMD/for-all/apply-to-all) or task-parallel (e.g. farm) way and it may exploit a distributed shared state, which survives to VPs lifespan. A module can nondeterministically accept from one or more input streams a number of input items, which may be decomposed in parts and used as function parameters to instantiate VPs according to the input and distribution rules specified in the parmod. The VPs may send items or parts of items onto the output streams, and these are gathered according to the output rules. The simple application in Fig. 1 a) includes three sequential modules (send1, send2, and recv) and one parmod (matrix_mul), which take two matrixes and give their product. The matrix_mul parmod is declared as follows:

`matrix_mul` is declared as follows:

```
1  parmod matrix_mul (input_stream long M1[N][N], long M2[N][N]
2                     output_stream long M3[N][N]) {
3    topology array [i:N][j:N] Pv;
4    attribute long A[N][N] scatter A[*ia][*ja] onto Pv[ia][ja];
5    attribute long B[N][N] scatter B[*ib][*jb] onto Pv[ib][jb];
6    stream long ris;
7    do input_section {
8      guard1: on , , M1 && M2 {
9        distribution M1[*i0][*j0] scatter to A[i0][j0];
10       distribution M2[*i1][*j1] scatter to B[i1][j1];
11     }   } while (true)
12   virtual_processes {
13     elab1 (in guard1 out ris) {
14       VP i, j {          f_mul (in A[i][], B[][j] output_stream ris);          }}}
15   output_section {
16     collects ris from ALL Pv[i][j] {
17       int elem; int Matrix_ris_[N][N];
18       AST_FOR_EACH(elem) {
19         Matrix_ris_[i][j]=elem;
20       }
21       assist_out(M3, Matrix_ris_);
22     }<>; } }
23 proc f_mul(in long A[N], long B[N] output_stream long Res)
24 $c++{ register long r=0;
25   for (register int k=0; k<N; ++k)
26     r += A[k]*B[k];
27   assist_out(Res,r); }c++$
```

The parmod exhibits a matrix of NxN VPs (SPMD, line 3). Once the two input matrixes are received (line 8), they are both scattered to the VPs which store them in the distributed shared matrixes A and B (lines 9–10) that has been previously declared (lines 4–5). Then, all elements of the result matrix C are computed in parallel (lines 12–14). Once all VPs completed the operation, a result matrix is collected from the distributed matrix C and sent into the output stream (lines 15–22). The code of sequential modules is not shown for the sake of brevity.

It is worth pointing out that the ASSIST programmer is not requested to code any low level detail of the application, especially those typical, cumbersome details of grid programming such as concurrency activities set up and mapping (firewalls, multi-tier networks with private address ranges), communication and synchronization protocols, dynamic QoS control under critical conditions (faulty or overloaded platforms and network links). The ASSIST compiler and its run-time support provide these features. More details on these features of ASSIST environment can be found in [4,3,2,5].

3. Module assembly via Web Services

The ASSIST compiler translates each module into a network of processes. Sequential modules are translated into sequential processes, while parmods are translated into a parametric (w.r.t. the parallelism degree) network of processes. Communications within a parmod are implemented by using ASSIST native communication libraries (relying on either plain TCP/IP or Globus); communication parameters and patterns are established and optimized at compile time. Communication among parmods (i.e. streams, depicted as arrows in Fig. 1) a) may be configured according to several standard protocols, among the others the native ASSIST streamlib (TCP/IP), and HTTP/SOAP. In the latter case the compiler automatically produces the code needed to attach a client/server pair to each stream ends. In particular:

- an XML file for each module describing the module interface, i.e. module name, input and output stream types.
- a gateway process for each input stream of each module. The gateway implements a gSOAP-based Web Service exporting a single asynchronous method [14]. The method invocation is meant to carry a stream item as method parameter. A suitable WSDL is generated.

- an *invoke* method for each output stream of each module. The invoke function is triggered by any output request on the output stream. Since the WSDL of the target Web Service will be known only at run-time, the implementation of the invoke is generated on the fly at run-time (after the wiring), and dynamically linked to the code.

- a *configure* method related to each invoke. It enables to overwrite the IP address of the Web Server the invoke connects to. This enable the independent deployment and run of modules and module run-time mobility (for fault tolerance or performance reasons).

In addition, the application graph is translated in an assembly of services described in a XML file. This information is used at launch time to wire modules one another. As example the following file describes the application in Fig. 1 a):

```
<ComponentConfiguration>
 <Assembly>
  <ComponentSection>
   <Component name="send1"  com="ws" kind="xml" file="./xmls/send1.xml"> </Component>
   <Component name="send2"  com="ws" kind="xml" file="./xmls/send2.xml"> </Component>
   <Component name="matrix_mul"  com="ws" kind="xml" file="./xmls/matrix_mul.xml"> </Component>
   <Component name="recv"  com="ws" kind="xml" file="./xmls/rec.xml"> </Component>
  </ComponentSection>
  <ConnectionSection>
   <Connection>
    <Output component="send1" interface="Matrix1"/>
    <Input component="matrix_mul" interface="Matrix1"/>
   </Connection>
   <Connection>
    <Output component="send2" interface="Matrix2"/>
    <Input component="matrix_mul" interface="Matrix2"/>
   </Connection>
   <Connection>
    <Output component="matrix_mul" interface="Matrix_ris"/>
    <Input component="recv" interface="Matrix"/>
   </Connection>
  </ConnectionSection>
 </Assembly>
</ComponentConfiguration>
```

The file does not directly contain any mapping of deployment information. The ASSIST launcher exploits heuristics to decide the mapping of modules onto platforms at launch time taking in account the kind of available platforms, user hints, application structure and grid current status [1,8]. Then, it relies on middleware (e.g. Globus) mechanisms to deploy and run each module and to wire them one another (see Fig. 1 b).

3.1. Web Services payback and overhead

The ASSIST programming environment, extended as described in the previous section, enables the programmer to describe an application as a composition of services. From an abstract viewpoint, such composite services exhibit the same functionality of the original ASSIST application. As matter of a fact, the only difference consists in protocol used to carry data from one module to another, which relies on HTTP/SOAP in the Web Service implementation. From a practical viewpoint, this may considerably ease application deployment since Web Services are usually allowed to cross site boundaries without any specific intervention on firewall configuration, and they are platform neutral. On the other hand, the overhead due to message marshalling and HTTP parsing may range from high to average depending on the specific marshalling technique.

As an example, Fig. 2 reports the comparison of the execution times achieved when running the same application (sketched in Fig. 2 left) using plain ASSIST (plain TCP/IP communications) or exploiting Web Services to run the *workers*, that is, the processes actually computing results out of (independent) input data items. The application processed 1K tasks (flowing from Dist to Coll, see Fig. 2), taking each ~ 65 milliseconds to be computed and requiring to serialize/marshall some

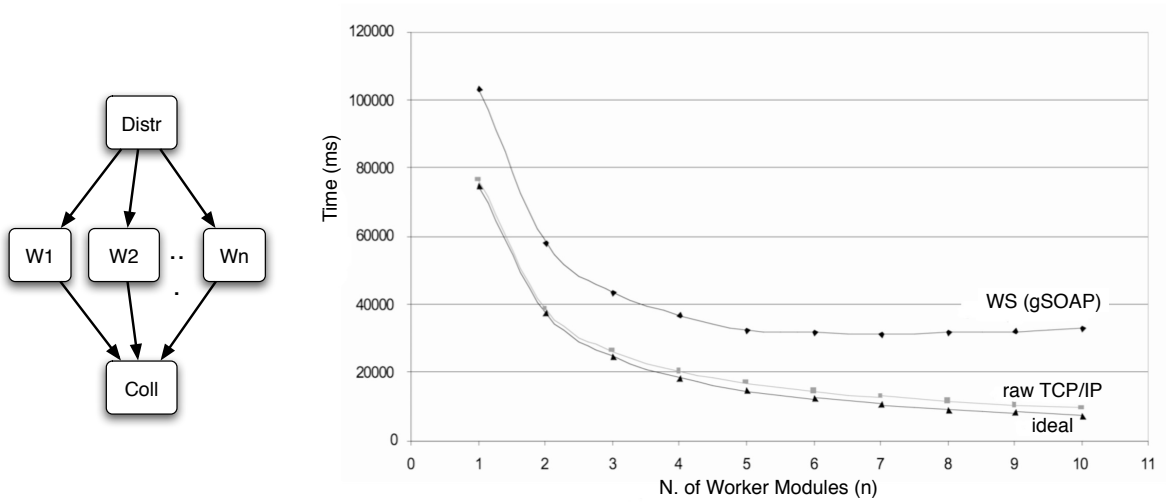


Figure 2. Plain TCP/IP and HTTP/SOAP protocols compared on a master-worker application.

Kbytes of data. The performance/scalability of the Web Service implementation is not extremely far from the one achieved using plain TCP/IP.

Differently from BPEL, the orchestration of services in an ASSIST application is fully decentralized. Since one-way streams connect ASSIST modules, all modules implement asynchronous Web Services. This solution may appear a bit unnatural when dealing with Web Services, as Web Services commonly exploit synchronous RPCs. However, this is also the underlying idea of the most promising proposals for efficient decentralized execution of BPEL-like workflows [17]. In particular the performance edge with respect to synchronous solution is likely to become a key factor for grid-aware high-performance applications. On the other hand, full interoperability with external, legacy Web Servers is very attractive. ASSIST modules support both one-way and RPC style HTTP/SOAP interaction as both client and server. Any ASSIST module (or a graph of them) can be wrapped and packaged into a so-called GRID.it component. As we shall see in the next section, a GRID.it component may inter-operate with other components both by means of one-way and RPC (use/provide) ports.

3.2. Component based grid programming: ASSIST and GRID.it

Some interesting, component based programming models have been proposed to be used in the grid context. In particular, the CORBA Component Model (CCM [13]) and the Common Component Architecture (CCA) component model [6] have been widely discussed in the grid context. Other models, coming from different experiences, such as JavaBeans, Web Services and Microsoft .NET are currently being considered in the field of grid programming although neither the web services model nor .NET can be properly called component models. In the context of the GRID.it project[2], our group introduced a fairly new component based programming model. Components can be either parallel or sequential. Legacy CCM components and WWW Web Services are assumed to be usable as sequential GRID.it components via proper wrapping. GRID.it components interact using three basic mechanisms, two inherited from previous component models and one which is brand new:

- classical use/provide ports are basically used to implement RPC-like component interaction.

[2]GRID.it is a three-year project, ending in 2005 that involves major Italian universities and research institutions [15]

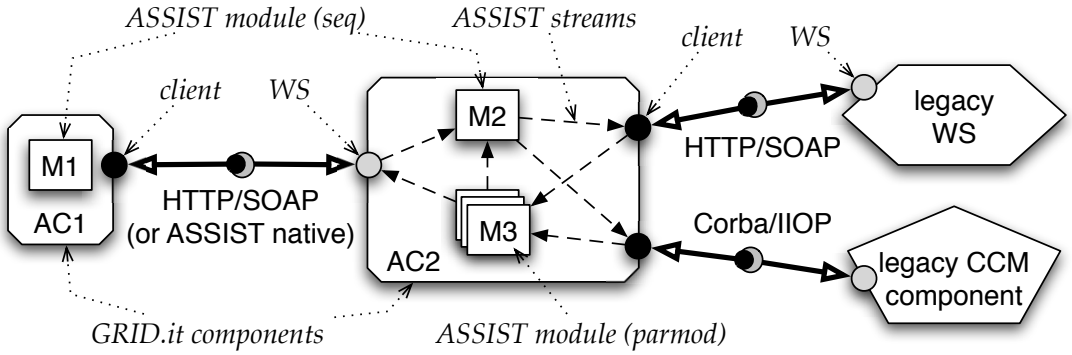


Figure 3. A grid application as an assembly of GRID.it components, CCM components, and legacy Web Server.

- events inherited from CCM, are basically used for component synchronization.
- data flow streams, a new mechanism that it is used to implement efficient, one-way data flow communication between components, are used to provide a way to transfer typed data items from one component (exporting a stream source interface) to another one (exporting the stream sink interface).

Even though data flow streams can be easily implemented in terms of either use/provide ports or events, they have been explicitly included in the set of primitive mechanism to enforce the concept that they provide optimized, high performance inter-component communication mechanisms. All these mechanisms are used to implement two distinct component interfaces:

- the functional interface exposing the component functional behavior to the other components. Using the mechanisms implemented in this interface a component can use the services provided by another component to actually compute a result
- the non-functional interface providing mechanisms that can be used to control the component behavior, that is, its execution features as well as its interaction with the underlying grid target architecture (not discussed in this paper, see [2,5]).

An ASSIST module (or a graph of modules) can be declared as a GRID.it component [2,3]. The ASSIST compiler automatically performs the wrapping of an ASSIST application graph into a GRID.it component membrane. As sketched in Fig. 3, a GRID.it component (e.g. AC1, AC2) may wrap any graph of ASSIST modules and it is characterized by provide (grey circles) and use ports (black circles). They may behave both in one-way/event and RPC-like manner, and may support several protocols, such ASSIST native, HTTP/SOAP , and CORBA/IIOP. These protocols enable the interoperability with other standard component models and paradigms.

3.3. GRID.it components and Web Services

All kind of GRID.it ports can be connected through HTTP/SOAP channels. Events and stream ports can be easily realized as described in section 3 since they are basically one-way channels. As RPC ports concern:

- a provide port behaves essentially as a (single method) stateful Web Service. When declaring a GRID.it component it is possible to bind a provide port to an ASSIST input stream and to

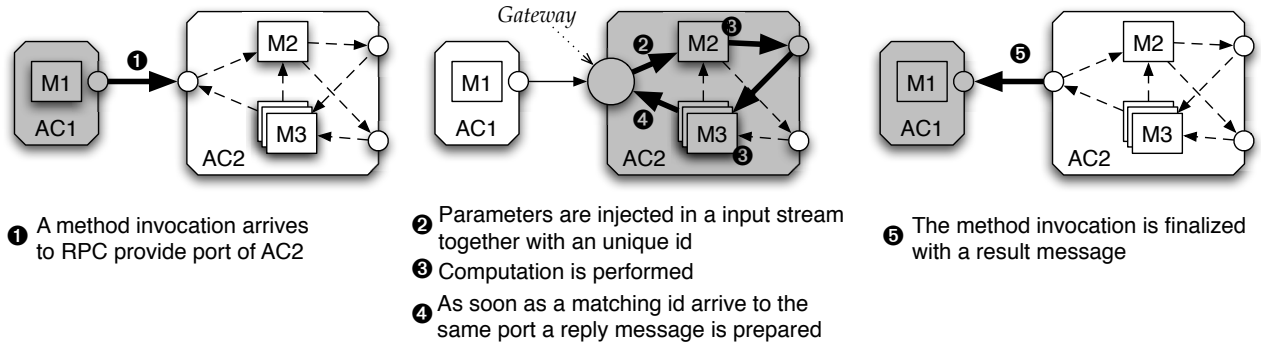


Figure 4. Evolution of a method call via Web Service in a GRID.it component.

an output stream within the component. An automatically generated gateway process, which is attached to the port and run the Web Server, collects method invocation from the provide port and injects the method parameter into the input stream bind to it. This data is properly colored with a unique identifier, which is carried along the entire ASSIST graph within the component. When the gateway collects a data item from the output stream matching the color of some ongoing method invocation in the port, a SOAP envelope is prepared and sent as service result. Coloring mechanism enable to distinguish possibly concurrent service requests on the same port. The succession of events is sketched in Fig 4.

- use ports are used as clients for Web Services. In this case is not possible to automatically generate all the code needed to invoke an external, possibly legacy Web Service because it depends on the service the port will be wired to (number and name of the methods, address, etc.). The GRID.it framework helps the programmer providing him with a proxy library whose entries are the stub methods for the remote Web Service. The library is generated starting from the target Web Service WSDL. The programmers have the responsibility to fulfill the stubs with the code needed to invoke the Web Service methods, and to place the calls to the stub methods within the sequential code of the ASSIST modules within the GRID.it components.

This solution has been already implemented and fully tested. It provides a bridge between ASSIST applications and GRID.it components and furtherly increases the interoperability opportunities deriving from the adoption of Web Service based mechanisms in the ASSIST/GRID.it grid programming environments.

4. Conclusions

The ASSIST environment provides a high-level programming alternative to the classic grid programming figure assuming that applications are built on top of grid middleware directly using/invoking the middleware functionalities at the user code level. As shown in previous works [4,3,2,5], ASSIST and GRID.it components can cope with many of the key issues of the grid programming while avoiding application programmers to entangle with the related cumbersome details such as software deployment. We shown that ASSIST modules can be composed as services with decentralized orchestration by using standard toolkits of Web Services development, and that all the code needed to turn ASSIST module interaction into Web Services can be automatically generated with no additional effort for the programmer. Also, we shown that ASSIST modules, wrapped as GRID.it components, can expose their functionalities as Web Services with any programmer intervention.

References

[1] M. Aldinucci and A. Benoit. Automatic mapping of ASSIST applications using process algebra. In *Proc. of HLPP2005: Intl. Workshop on High-Level Parallel Programming*, Warwick University, Coventry, UK, July 2005.

[2] M. Aldinucci, S. Campa, M. Coppola, M. Danelutto, D. Laforenza, D. Puppin, L. Scarponi, M. Vanneschi, and C. Zoccolo. Components for high performance Grid programming in Grid.it. In V. Getov and T. Kielmann, editors, *Proc. of the Workshop on Component Models and Systems for Grid Applications*, CoreGRID series. Springer Verlag, January 2005.

[3] M. Aldinucci, M. Coppola, S. Campa, M. Danelutto, M. Vanneschi, and C. Zoccolo. Structured implementation of component based grid programming environments. In V. Getov, D. Laforenza, and A. Reinefeld, editors, *Future Generation Grids*, CoreGRID series. Springer Verlag, November 2005.

[4] M. Aldinucci, M. Coppola, M. Danelutto, M. Vanneschi, and C. Zoccolo. ASSIST as a research framework for high-performance Grid programming environments. In J. C. Cunha and O. F. Rana, editors, *Grid Computing: Software environments and Tools*. Springer Verlag, January 2006.

[5] M. Aldinucci, A. Petrocelli, E. Pistoletti, M. Torquati, M. Vanneschi, L. Veraldi, and C. Zoccolo. Dynamic reconfiguration of grid-aware applications in ASSIST. In *11th Intl Euro-Par 2005: Parallel and Distributed Computing*, volume 3648 of *LNCS*, pages 771–781, Lisboa, Portugal, August 2005. Springer Verlag.

[6] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a common component architecture for high performance scientific computing. In *Proc. of the 8th Intl. Symposium on High Performance Distributed Computing (HPDC'99)*, 1999.

[7] OASIS Technical Committee. WSRF: The Web Service Resource Framework Globus web site. http://www.oasis-open.org/committees/wsrf.

[8] M. Danelutto, M. Vanneschi, C. Zoccolo, N. Tonellotto, S. Orlando, R. Baraglia, T. Fagni, D. Laforenza, and A. Paccosi. HPC application execution on grid. In *Dagstuhl Seminar Future Generation Grid 2004*, CoreGRID series. Springer-Verlag, 2005. To appear.

[9] J. Dünnweber, S. Gorlatch, M. Aldinucci, S. Campa, and M. Danelutto. Behavior customization of parallel components application programming. Technical Report TR-0002, Institute on Programming Model, CoreGRID - Network of Excellence, April 2005.

[10] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. The physiology of the grid. http://www.globus.org/research/papers/, June 2002.

[11] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: Enabling scalable virtual organization. *The Intl. Journal of High Performance Computing Applications*, 15(3):200–222, Fall 2001.

[12] S. Graham, S. Simeonov, T. Boubez, G. Daniels, D. Davis, Y. Nakamura, and R. Neyama. *Building Web Services with Java: Making Sense of XML, SOAP, WSDL, and UDDI*. SAMS, 2001.

[13] Object Management Group. Corba component model version 3.0 specification. http://www.omg.org, September 2002.

[14] gSOAP C++ Web Services home page. http://www.cs.fsu.edu/∼engelen/soap.html, 2003.

[15] The GRID.it home page. http://www.grid.it.

[16] K. Kennedy, M. Mazina, J. Mellor-Crummey, K. Cooper, L. Torczon, F. Berman, A. Chien, H. Dail, O. Sievert, D. Angulo, I. Foster, D. Gannon, L. Johnsson, C. Kesselman, R. Aydt, D. Reed, J. Dongarra, S. Vadhiyar, and R. Wolski. Toward a framework for preparing and executing adaptive Grid programs. In *Proc. of NSF Next Generation Systems Program Workshop (IPDPS 2002)*, 2002.

[17] M. G. Nanda, S. Chandra, and V. Sarkar. Decentralizing execution of composite web services. In *Proc. of Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Vancouver, B.C., Canada, October 2004. ACM.

[18] M. Vanneschi. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing*, 28(12):1709–1732, December 2002.

Performance Evaluation & Analysis

Performance Analysis of Parallel Applications with KappaPI 2*

J. Jorba[a], T. Margalef[a], E. Luque[a]

[a]Computer Architecture and Operating Systems Department, Universidad Autonoma de Barcelona, 08193 Bellaterra, Spain

1. Introduction

Performance is a crucial issue in parallel/distributed applications. Designers and developers expect their applications to reach certain performance indexes to meet the expectations of high performance computing systems. Therefore, parallel application developers are obliged to not only to analyze the application itself, but also the software layers involved and the distributed system behavior, in order to detect any performance bottlenecks that appear during the execution of the application, determine their causes and modify the application to overcome such bottlenecks. So, performance analysis becomes a highly significant task in the development of parallel/distributed applications.

However, there is still a lack of geuinely useful tools and the most popular approach to carrying out performance analysis is the use of visualization tools [1–3] that offer to the developer a set of views of the application execution. The analysis of these views is a difficult and time consuming task that requires a high degree of expertise of the user.

To tackle all these problems, user-friendly tools should be available. In this context several automatic performance analysis tools have been developed, Expert [4], Scalea [5]. These tools take a trace file from the execution of the application and try to detect performance bottlenecks using certain performance properties. The post-mortem approach has the advantage of being able to consider all detailed information gathered during application execution and, moreover, the analysis phase does not introduce any overhead in the application execution. Using this approach it is possible to perform a comprehensive analysis, identifying the most important performance bottlenecks.

To identify the performance bottlenecks these tools use certain performance property specification derived from the APART Specification Language (ASL) [6]. This specification language makes it possible to specify the set of events in a trace file that determines the appearance of a particular performance bottleneck.

These tools have been tested in a wide range of applications that include synthetic applications specially designed to present certain performance bottlenecks (APART Test Suite ATS [7]). The results show that these tools detect performance bottlenecks fairly efficiently. However, the main limitation of these tools is that they detect some performance bottlenecks, but are not able to relate the performance bottleneck with the application's code and do not provide any hints to help the developer overcome them. So, a more profound analysis and more comprehensive and focused information must be provided to the developer.

A more ambitious tool is KappaPI (Knowledge-based Automatic Parallel Program Analyzer for Performance Improvement) [8,9]. This tool takes trace files from PVM applications, detects performance bottlenecks by determining inactivity time intervals, and determine their causes by applying certain inference rules, relate them to the application source code and finally suggest certain recommendations to the user. One of the main disadvantages of this tool is that the performance knowledge

*This work was supported by the MCyT under contract number TIN 2004-03388 and partially funded by the Generalitat de Catalunya Grup de recerca consolidat 2001-SGR-00218.

(a small set of performance problems), is directly coded in the kernel of the tool. This means that it is not possible to modify some performance bottleneck or define new performance bottlenecks without recoding the tool kernel itself.

Keeping in mind the same goal of providing useful hints to the application developer, a new tool, KappaPI 2, has been designed. In KappaPI 2, the approach is based in the specification of performance knowledge as input data for the tool. This knowledge represents a set of parallel performance bottleneck patterns that can be found in the execution of parallel/distributed applications. The specification is made in XML syntax, similar to the APART specification Language (ASL) [6]. The performance bottlenecks included in the current implementation are Late_Sender, Late_Receiver, Messages_In_Wrong_Order, Multiple_Output, Blocked_Barrier, and so on [6].

The rest of the paper is organized as follows. Section 2 surveys related work on automatic performance tools. Section 3 shows basic concepts and general structure of the KappaPI 2 tool. Section 4 summarizes the idea of structural bottleneck specification. Section 5 presents the mechanisms (and some low level details) of bottleneck detection and classification. In section 6 we show the internals of cause analysis and the related analysis of application source code. Section 7 provides some experimental results of the tool with a testbed suite of benchmarks and real applications. Finally, section 8 summarizes our conclusions.

2. Related work

Existing tools related to KappaPI 2 include several automatic performance tools, such as the first version of KappaPI [9], Expert [4] and Scalea [5].

In the first version of KappaPI, detection of performance bottlenecks was focused on idle intervals in the global computation affecting the largest number of tasks. Processor efficiency was used to measure the execution quality, and idle processor intervals represented performance bottlenecks. The tool examined the intervals to find the causes of the inefficiency. The tool had a closed hard-coded set of bottlenecks, and no mechanisms for new bottleneck specification was included. The same limitation also affects root cause analysis.

The Expert [4,12] tool allows the user to specify performance properties using a script language based on an internal API. This API makes it possible to examine the trace and to look for relationship between events. Expert summarizes the indexes of each bottleneck defined in its list of bottlenecks. Expert tries to provide an answer to the question of where is the application spends time. It summarizes the performance bottlenecks found and accumulates their times to compare their impact on the total execution time. Main differences between KappaPI 2 and Expert are: a) Specification as bottleneck related events from a structural point of view as opposed to functional programming (in a shell scripting language). b) Expert specification is based on a trace API, that the user needs to know in order to specify the bottleneck properties, in Kappa PI 2 this use of the trace is only internal. KappaPI 2 offers abstraction mechanisms from trace formats and environments: we can use different tracers in different environments (PVM, MPI). c) Expert doesn't offer techniques for source code analysis, or hints to help the final user to improve their application. d) In Kappa PI 2 an additional level of specification is added for bottleneck analysis causes. The user can provide knowledge about bottlenecks and their analysis process.

Scalea [5] is used with the Aksum tool [13] for multi-experiment performance analysis. This tool uses an interface called JavaPSL to specify the performance properties by using syntax and semantic rules of the Java programming language. The user can specify new properties and formats without changing the implementation of the search tool that uses JavaPSL API.

3. KappaPI 2

Our goal is to design and implement an automatic performance analysis tool that has the following features:

- Performance knowledge specification: Independent specification mechanisms to introduce new performance bottlenecks.
- The performance bottleneck detection engine must read the performance knowledge specification.
- Independence from background message passing system: The tool builds abstract entities that are independent from the particular trace file format or the message passing primitives.
- Relate bottlenecks to the source code of the application: a set of quick parsers to search for dependences in the source code must be included to determine why the bottleneck appears.

In KappaPI 2 (figure 1), the first step is to execute the application under the control of a tracing tool (for PVM or MPI environments) that captures all the events related to the message passing primitives that occur while carrying out the application. Our tool uses the trace and performance bottleneck knowledge base as inputs to detect the performance bottleneck patterns defined from a structural point of view. After, it sorts the performance bottlenecks it has found in accordance with certain indexes. It carries out a bottleneck cause analysis, based on the application source code analysis, and finally provides a set of recommendations to the user, indicating how to modify the source code to overcome the detected bottlenecks.

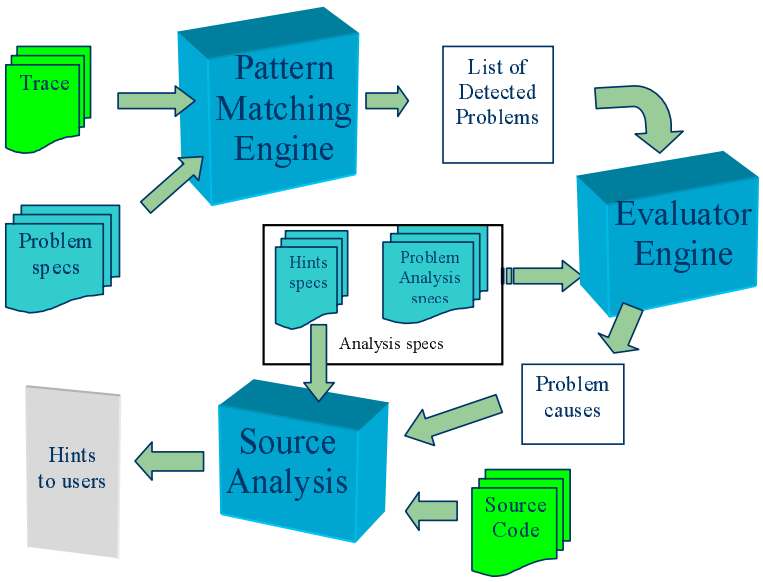


Figure 1. Module architecture of the KappaPI 2

4. Bottleneck Specification

In KappaPI 2, bottlenecks [15] are specified using a structural point of view. Each bottleneck is described as a set of events (which need to be found in the trace) and their relations. The specification is based on the compound event concept in the ASL language [6], in our case the concept is translated to an XML syntax to specify the bottleneck from a structural point of view.

Each bottleneck has an initial event (Root event), followed by event instances involved, and certain constraints on time and location, and required computations to evaluate the impact of the bottleneck.

This bottleneck specification avoid hard-coded knowledge in the tool, and makes the tool open to the incorporation of any new knowledge developed about new bottlenecks. Another interesting point concerns tool personalization: one set of bottlenecks define one instance of the performance tool. For example, we can only specify p2p communications or collective bottlenecks, or only search for one particular mix of bottlenecks. This defines an interesting experimentation framework for performance analysis, to test different sets of knowledge as input.

5. Bottleneck Detection and Classification

Since KappaPI 2 uses a post-mortem approach it is necessary to execute the application with a tracer tool that collects all the events related to the message passing primitives that occurs during the execution of the application and stores them in a trace file.

The first step to carry out the performance analysis is to read the specification of the performance bottleneck patterns and to create a classification tree. Each bottleneck is show in this tree as a path. Matching the events found in the trace file in the classification tree defines the detection of a bottleneck. In the bottleneck path, each node has one event instance of the bottleneck, the position in the path defines the order of the events in the trace (this order is built by means of constraints specified in the bottleneck).

In the bottleneck classification tree there are three different types of nodes: 1) root as initial node with root event instance; 2)medium nodes as intermediate bottleneck event instance; and 3) end nodes when the match has been completed, and where some computations (included in specification) are evaluated.

But this type of end node appears as two different versions, final and leaf node. The difference is related to the shared partial path between several bottlenecks. For example a simple case to see (shown in figure 2), involves a LateSender and BlockedSender shared path: a BlockedSender can initially be seen as a LateSender. If we reach the final LateSender node, we can consider that a bottleneck has been found, but we need to consider not stopping at that point, if more events related to BlockSender appear afterwards. In this case we need to maintain this partial (LateSender) match until we can test whether it is possible to continue the path or not. This is the case of final node, in case we have arrived at the final node of blocked sender, we consider this node (if there is no other path from this node) a leaf node, and it can directly be incorporated in the list of bottlenecks found.

Once the classification tree has been created and loaded in memory, KappaPI 2 takes the events from the trace file and matches them in the classification tree to detect the performance bottlenecks that occurred during the execution of the application.

During the matching of read bottleneck specification (organized as the classification tree) and the event stream of the application, we need to consider the aliasing problems that can occur. This aliasing is caused by the use of variables representing the events (their names). In each bottleneck the user, for the description, use symbolic names for the events, their constraints and the calculations associated with the bottleneck. In the matching process between the application event stream and

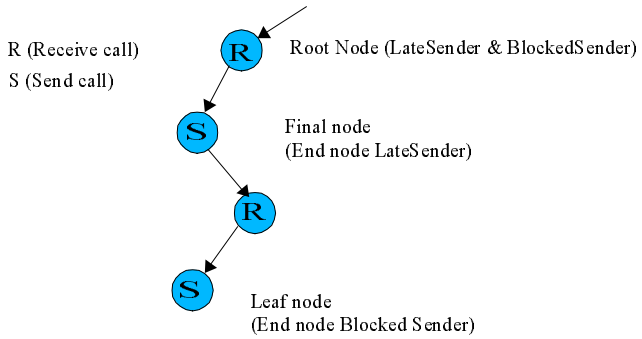


Figure 2. Shared tree path between LateSender and BlockedSender

the classification tree, an attempt is made to associate each event with one name. If a match occurs, the event is included in a partial bottleneck match, and must be included in a symbol table with the symbolic pair {event symbolic name, event matched}. This aliasing is not an easy process, because each bottleneck specification has one set of events, in which some are ordered casually, and others without any specific causal constraints (they are required events, but they have not time constraints). This set of events (without time constraints) generate combinations of different possible aliasings.

Once the bottleneck table has been created, classification indexes are used to evaluate the impact of the bottleneck. The goal is to filter only a small set of problems, which provokes the greatest inefficiency impact. This impact of the bottleneck is evaluated in terms of locality (causes inefficiencies in one reduced set of tasks, or in the global application), and temporal impact (degree of time impact). Another aspect to check is the repetition of one bottleneck, or the overlapping of different bottlenecks, and their resulting impact.

After classification, in the analysis phase we try to find what causes of the bottleneck to occur.

6. Cause Analysis and Source Code

Once the performance bottlenecks have been detected and classified, the next steps are the determination of the actual causes of the bottlenecks and the construction of certain hints to the user. The determination of the actual cause of the bottleneck usually implies some source code analysis, to detect for example, any data dependences. To carry out this phase properly it is necessary for the events in the trace to include information concerning the source code line that has been generated by each one of them. In our case we developed a tracer based on dynamic instrumentation that uses the DyninstAPI [10], which provides all the required information for MPI applications. In the case of PVM applications we use the TapePVM tracer.

The actual analysis that must be performed done depends on the performance bottleneck detected. Therefore, this analysis and the construction of hints, are also included in specification files linked to the bottleneck specification file that is loaded by the KappaPI 2 kernel.

This cause analysis concept adds an extra level of specification, providing a second level of openness in the KappaPI 2 tool, and making the tool more generic for incorporating bottleneck knowledge.

In the specification of cause analysis, we relate a bottleneck to a series of cases to be tested. Each case can include certain relationships between source code that must be verified.

These relations, and tests of source code are specified as calls to a small source code parsing API,

which are used in the specification of analysis cases. The relation is expressed in terms of certain basic parsing techniques, such as testing dependence of variables, function calls, parameters in use, etc. These analyses are produced by a set of quick parsers with a well defined function (parsing API) to search for a hypothesis in the source code. For example: Can that code (or call) be moved up or down? Does the code have any data dependences in the actual block? ... and so on.

The source code analysis is carried out in two phases: The first is related to code scope (block analysis), through the use of a code representation based on SIR [11] representation. It makes it possible to detect the region structure of the message passing calls analyzed, relating the call with their scope(conditional, loop, ...). And the second implies the request to verify a relation or a dependence test to apply at the region (block and calls) of source code involved in the bottleneck case tested.

The analysis of the relations and scope of the code produces the input needed to check the use cases, and to define the final suggestions to be made to the user depending on the relations found in the source code. The hints for users are based on templates defined in each case tested. The hint template is filled, when an analysis case is matched, with associated event data and source code.

7. Experiments

We carried out a series of experiments to validate the different phases of our tool: detection, classification, cause analysis (with source code analysis) and final hint suggestions to the user.

Different tests were made, including synthetic benchmarks, performance available benchmarks, and some real applications. The test suite is a set of MPI and PVM applications. We use some synthetic benchmarks based on the ATS (APART Test suite)[7] which makes it possible to easily create benchmarks with a specific set of known bottlenecks (for example LS, LR and BS cases). Other tests and classic benchmark were used such as pingpong, or IS (Integer Sort) from NAS benchmarks. The Intel Benchmark suite (IMB) includes various benchmarks with different p2p communication calls. As real applications, we used two different applications related to ecological environments: Xfire [14], a simulator of forest fire propagation. And a fishtank simulator (called FishSchool). Xfire is a C PVM based application, and FishSchool is a C++ MPI application.

Table 1 shows the mix of experiments, with data related with the number of bottlenecks found, and the impact of main bottlenecks on the global application time:

Benchmark	bottlenecks found	Idle time (% total time)
Apart LS (mpi)	1LS	54%
Apart LR (mpi)	0	0%
Apart BS (mpi)	1BS	45%
PvmLR (pvm)	1LR	40%
PingPong (pvm)	5LS	main 2LS 89%
NAS IS (pvm)	27LS, 18BS	main 2LS, 1BS 20%
IMB-MPI1 p2p (mpi)	3LS, 1BB	main 3LS 32%
Xfire (pvm)	17LS, 6BS	main 2LS 51%
Fish school (mpi)	15LS, 2BB	main BB 20%

Table 1
Summary of application test suite

The first series of experiments, with the APART test suite, are used as validation of the detection phase (of KappaPI 2 tool), for each individual bottleneck.

To see all phases of the tool, in a simple case, we select the LateSender case in MPI (Apart LS) builded by using the APART test suite:

First, the tool detects this bottleneck, based on its specification [15] and its representation as a path, in the classification tree, of two events with a receive (mpi call) as the root event, and the one related send event. In this benchmark case, the LS bottleneck is the only one found. In the matching process within the tree, two events from the application trace are aliased to the events in the description, and their information was collected by relating the events to the tasks, times, and source code position. In the final table of the detection phase, one entry was found about one late sender between two tasks, and a computed idle time in the receiver task.

In the cause analysis, we need to test certain relationships in the task source code involving bottleneck cases, in the sender case we can test whether it is possible to move the send call forward to avoid possible data dependences, or in the case of the receiver, whether it is possible to postpone the receive call, moving some computations not related to dependences with the receive call forward. If those moves are not possible, we can suggest a different mapping of tasks, to provide a mapping for a sender task that permits a quicker sender task.

In the example analized, Apart LS, the KappaPI 2 tool detects a LateSender bottleneck, related to two events a delayed MPI_Send, and a MPI_Recv and their lines(4,11) in source code (see code extracts):

Part of Apart LS code

```
1   for (n=1;n<=reps;n++) {
2     do_work(2);
3     MPI_Send(message, strlen(message)+1, MPI_CHAR,
4              dest, tag, MPI_COMM_WORLD);
5   }
6 }
7 else {
8   source=1;
9   for (n=1;n<=reps;n++) {
10    MPI_Recv(message, 100, MPI_CHAR, source, tag,
11             MPI_COMM_WORLD, &status);
12    do_work(1);
13  }
14 }
```

In the analysis phase it needs to test the case if the send call, can be move forward. In SIR code representation it sees MPI_Send on a for loop, and after analysis of block code does not detect any dependences (do_work() is a call to simulate CPU computing time, in Apart test suite API). As result, the tool provides a hint to the user, with the suggestion of move send code as the first line of the for loop (to go ahead do_work() call). This hint produces a benefit of 25% in execution total time of the benchmark.

8. Conclusions

KappaPI 2 has been tested on a wide range of applications, including the APART Test Suite applications and also some real applications. Quantitative measurements obtained from the set of benchmarks, on the proposed tool architecture, corroborate that the detection of parallel performance problems based on structural patterns, the use of an open tool to incorporate knowledge in a highly

flexible form, and the generation of suggestions directed to the developer is a feasible approach that helps the performance tuning process significantly.

Our research is addressed at the design of an open solution for a general automatic performance tool, which can accept the state of the art in parallel performance knowledge as input, and is used to improve the process of the tuning of the parallel/distributed applications by suggestions aimed at an non-expert user.

We have discussed a new automatic performance analysis tool oriented toward the end user to avoid the high degree of expertise needed to improve message-passing applications. Our KappaPI 2 tool is open to the introduction of new models for performance bottlenecks. It is able to make suggestions about the source code to improve the application execution time and therefore avoid performance bottlenecks. The experiments carried out show that the tool detects specified bottlenecks which can be related to the source code and provide hints to help the user avoid them.

References

[1] W.E. Nagel, A. Arnold, M. Weber, H.C. Hoppe, K. Solchenbach: VAMPIR: Visualization and Analysis of MPI Resources. Supercomputer 63, vol XII, number 1, Jan. 1996

[2] T. Cortes, V. Pillet, J. Labarta, S. Girona: Paraver: A tool to visualize and analyze parallel code. In WoTUG-18, pages 17-31, Manchester, April 1995.

[3] L. De Rose, Y. Zhang, D.A. Reed: SvPablo: A Multilanguage Performance Analysis system. Lecture Notes in Computer Science, 1469:352-99, 1998

[4] F. Wolf, B. Mohr, J. Dongarra, S Moore: Efficient Pattern Search in Large Traces Through Succesive Refinement. Euro-Par 2004, LNCS 3149, 2004.

[5] HL. Truong, T. Fahringer, G. Madsen, AD. Malony, HMoritsch, S. Shende: On using SCALEA for Performance Analysis of Distributed and Parall el Programs. Supercomputing 2001 Conference (SC2001), Denver, Colorado, USA. November 10-16,2001

[6] T. Fahringer, M. Gerndt, G. Riley, J. Larsson: Specification of Performance problems in MPI Programs with ASL. Proceedings of ICPP, pp. 51-58. 2000.

[7] M. Gerndt, B. Mohr, JL. Trff: Evaluating OpenMP Performance Analysis Tools with the APART Test Suite. Euro-Par 2004, LNCS 3149, 2004.

[8] A. Espinosa, T. Margalef, E. Luque: Automatic Performance Evaluation of Parallel Programs. IEEE Proceedings of the 6th Euromicro Workshop on Parallel and Distributed Processing. Jan. 1998.

[9] A. Espinosa, T. Margalef, E. Luque: Automatic Performance Analysis of PVM applications. EuroPVM/MPI 2000, LNCS 1908, pp. 47-55. 2000.

[10] J.K. Hollingsworth, B. Buck: DyninstAPI Programmers Guide. Release 3.0. University of Maryland, January 2002.

[11] T. Fahringer. Towards a Standardized Intermediate Program Representation (SIR) for Performance Analysis. (Slides at http://www.fz-juelich.de/apart/sc03), 5th International APART workshop, SC2003, Nov 2003.

[12] F. Wolf, B. Mohr: Automatic Performance Analysis of MPI Applications Based on Event Traces. In EuroPar 2000, LNCS, 1900, pp123-132, 2000.

[13] C. Seragiotto, M. Geisller, et al: On Using Aksum for Semi-Automatically Searching of Performance Problems in Parallel and Distributed Programs. Procs. Of 11th Euromicro Conference on Parallel Distributed and Network based Processing (PDP) 2003.

[14] J. Jorba, T. Margalef, E. Luque, J. Andre, D. Viegas: Application of Parallel Computing to the Simulation of Forest Fire Propagation. Proceedings of International Conference in Forest Fire Propagation, Vol 1, pp 891-900, Portugal, Nov.1998.

[15] J. Jorba, T. Margalef, E. Luque: Automatic Performance Analysis of Message Passing Applications using the KappaPI 2 tool. EuroPVM/MPI 2005, LNCS 3666, September 2005.

A Comparative Evaluation of Two Techniques for Predicting the Performance of Dynamic Enterprise Systems

David A. Bacigalupo [a], Stephen A. Jarvis[a], Ligang He[a], Daniel P. Spooner[a], Denis Pelych[b] and Graham R. Nudd[a]
daveb@dcs.warwick.ac.uk

[a]Department of Computer Science, University of Warwick, Coventry CV4 7AL, UK

[b]The National B2B Centre, Coventry CV4 7AL, UK

This paper is concerned with predicting the response times an enterprise information system would provide on new server architectures. These predictions can allow a workload to be transferred onto new servers whilst maintaining quality of service levels. Two common techniques are solving queuing models and extrapolating from previously gathered performance data. The dynamic recalibration of a layered queuing model and a historical model are investigated experimentally using an established distributed enterprise benchmark. The conclusions provide guidelines as to how to select an appropriate technique, including how to dynamically calibrate each model at a low overhead. Using these guidelines it is shown that both techniques can make low overhead predictions for new server architectures at a good level of predictive accuracy (typically over 80%) [1].

1. Introduction

It has been shown that response time predictions can enhance the workload and resource management of enterprise information systems [1,2]. Two common approaches used in the literature for making these response time predictions are extrapolating from historical performance data and solving queuing network models. Examples of the first approach include the use of both coarse [3] and fine [1] grained historical performance data. The former involves recording workload information and operating system/database load metrics, and the later involves recording the historical usage of each machine's CPU, memory and IO resources by different classes of workload. Another example of this approach is being developed in the High Performance Systems Group at the University of Warwick [4]. This historical technique has been implemented as a tool called HYDRA which has been applied to both distributed enterprise [4,5] and business-to-business [6] applications. It is differentiated from other historical modelling work by its focus on simplifying the process of analysing any historical data so as to extract the small number of trends that will be most useful to a resource management system.

Examples of the queuing modelling approach include [7,8,2] and the layered queuing technique, as implemented in the layered queuing network solver (LQNS) [9]. The layered queuing technique is of particular interest and will be examined further in this paper as: it explicitly models the tiers of servers found in this class of application, and it has been applied to a range of distributed systems (i.e. [10]) including the distributed enterprise benchmark used in this paper [11].

It is important to compare the effectiveness of different approaches for modelling enterprise information systems so practitioners can make an informed choice when designing prediction-enhanced workload and resource management systems. However, although there have been comparisons of

[1]The work is sponsored in part by the EPSRC (contract no. GR/S03058/01 and GR/R47424/01), the NASA AMES Research Center administered by USARDSG (contract no. N68171-01-C-9012) and IBM UK Ltd.

different performance prediction approaches using enterprise information systems, there have been few quantitative comparisons of the two approaches on a single enterprise application. For example in [10] a layered queuing model of a distributed database system is created and compared to a markov chain-based queuing model of the system. In [9] the layered queuing technique is compared more generally to other performance modelling techniques. Another recognised queuing technique which has been applied to similar applications is described in [7] and compared with the layered queuing technique. However none of these papers include a comparison with a historical model of the same application. The historical prediction technique described in [3] is applied to a web-based stock trading application (Microsoft FMStocks) and compared to a queuing modelling approach. However a queuing network model is not created of the application.

This paper describes such a comparison for dynamic enterprise information systems. These are systems which must continually adapt to changes in the workload, system configuration (i.e. monitoring and logging policies) and available servers. This may involve workload managers acquiring new servers for which only a small number of benchmarks have been run (i.e. to determine request processing speed). We investigate how the prediction models can be rapidly recalibrated with low overheads on an established server, whilst still obtaining enough data to make accurate predictions on new server architectures. This allows the models to be recalibrated prior to making real-time workload management decisions, and removes the need to model the workload and system configuration variables that change less frequently at runtime. This has a number of advantages when predicting the performance of dynamic systems including: i.) a reduction in model complexity which can dramatically improve the responsiveness of predictions; and ii.) removing the need to consider some variables which may be complex to measure and model (such as the complexity of processing the data in the database for each service class in the workload).

The comparison involves comparing the HYDRA historical technique and the layered queuing technique using a distributed enterprise application benchmark. The contributions of the work are to: i.) investigate the dynamic recalibration of these models experimentally; ii.) provide guidelines for making accurate predictions using both techniques at a low overhead; and iii.) comparatively evaluate the two techniques to help users decide which technique to use. The IBM Websphere middleware [12] is selected as the platform on which the benchmark will be run as it is a common choice for distributed enterprise applications. The IBM Performance Benchmark Sample 'Trade' [13] is selected as it is the main distributed enterprise application benchmark for the Websphere platform.

This remainder of this paper is structured as follows: defining a system model, sample workload and experimental setup (see section 2); investigating the recalibration overheads and accuracies of the two techniques (see sections 3 and 4); and describing the recalibration guidelines and comparative evaluation (see section 5).

2. System Model and Sample Workload

Based on established work (i.e. [11,14]) the enterprise information system is modelled as a tier of application servers accessing a single database server. Application servers may have heterogeneous server architectures. Based on the queuing network in the Websphere e-Business platform: a single first in first out (FIFO) waiting queue is used by each application server; the database server has one FIFO queue per application server; and both types of server can process multiple requests concurrently via time-sharing. The workload consists of clients (divided into service classes) which send requests to the system. The workload manager adjusts the routing of the incoming requests to the application servers which may involve acquiring new servers from another system or from a

computational grid.

The sample workload is as follows. A service class is created for 'browse' users with the next operation (i.e. buy/sell/quote etc) called by a client being randomly selected, with probabilities defined as part of the Trade benchmark. For simplicity, the typical workload is defined as all browse clients. A service class is created for 'buy' users which involves clients making an average of 10 buy requests. 'No. of clients and the mean client think-time' is used as the primary measure of the workload from a service class. Using number of clients (as opposed to a static arrival rate definition) to represent the amount of workload is common when modelling enterprise information systems (i.e. [8,11]). This is because it explicitly models the fact that the time a request from a client arrives is not independent of the response times of previous requests, so as the load increases the rate at which clients send requests decreases. Think-times are exponentially distributed with a mean of 7 seconds for all service classes as recommended by IBM for Trade clients, although heterogeneous think-times are supported by both techniques.

The experimental setup contains 3 application servers running Websphere Application Server (v4.0.1). Under the typical workload the max throughputs of the new 'slow' server $AppServ_S$ (P3 450Mhz), the established 'fast' server $AppServ_F$ (P4 1.8Ghz) and the established 'very fast' server $AppServ_{VF}$ (P4 2.66Ghz) are found to be 86, 186 and 320 requests/second respectively under the typical workload. The database is DB2 7.2 (on an Athlon 1.4Ghz) and 250 clients are simulated by each workload generator (P4 1.8Ghz). All servers run Windows 2000, have at least 512MB RAM and are connected via a 100Mbps switch.

3. The Layered Queuing Technique

A layered queuing performance model explicitly defines an application's queuing network. An approximate solution to the model can then be generated automatically, using the layered queuing network solver (LQNS). The solution strategy involves dividing the queues into layers corresponding to the tiers of servers in the system model, generating an initial solution and then iterating backwards and forwards through the layers solving the queues in each layer by mean value analysis and propagating the result to the next layer until the solutions converge. Performance metrics generated include response times, throughputs and utilisation information for each service class. A detailed description of the layered queuing technique can be found in [9].

A layered queuing model is created with application server, database server and database server disk layers, each layer containing a queue and a processor. The application server disk is not modelled as the Trade applications utilisation of this resource is found to be almost 0 during normal operation. Workload parameters (per service class) are: the number of clients, the mean processing times on each processor, and the average number of database requests per application server request. Processing times are assumed to be exponentially distributed. Queue parameters include the maximum number of requests each processor can process at the same time via time-sharing. Communication time is represented as a constant delay which is calibrated by subtracting the predicted response time from the actual response time at a small number of clients (250 in the experimental setup) on an established server. In the experimental setup the application server, database server and database disk can process 50, 20 and 1 requests at the same time, respectively. And it is found that the buy service class makes 2 database requests, and the browse service class makes 1.14 database requests on average.

The service class mean processing times are calculated during recalibration by taking an established server off-line and sending a workload consisting only of that service class. This overcomes the difficulties that have been found measuring mean processing times (without queuing delay) of

multiple service classes, in real system environments [14]. The mean processing time is then calculated by dividing the CPU/disk usage for the server/database disk, respectively by the throughput (in requests/second). Calculating the service class mean processing times on a new server architecture involves multiplying the mean processing times on the established server by the established/new server request processing speed ratio.

Experiments are conducted to examine the predictive accuracy and resource usage overhead when calibrating the request processing times under different amounts of background workload. The typical workload is calibrated on an established server with a maximum throughput of 213 requests/second. Each test run involves activating the required number of clients and waiting 1 minute for the system to reach a steady state. The %CPU/disk usage samples are then recorded for a period of 1 minute along with the mean throughput of the servers during the minute. The sampling interval is set at 6 seconds so the increase in the %CPU/disk usage is no more than 5%. Predictive accuracy is defined as:

$$accuracy = \frac{|predicted\_value - measured\_value|}{measured\_value} \times 100 \quad (1)$$

It is found that as the number of clients is increased the mean processing time does not remain constant. Instead it decreases (from 5.6ms at 63 clients) to a minimum of 4.3ms at 750 clients and 45% application server CPU usage, and then increases (to 4.7ms at 2250 clients). This pattern is explained as follows. The higher mean processing times at smaller number of clients are due to the larger system and JVM overhead (i.e. for garbage collection) per request. The higher mean processing times at larger numbers of clients are due to the overhead of running a larger number of threads (as Websphere terminates threads that are not needed, at lighter loads). At intermediate numbers of clients these overheads are less significant resulting in lower mean processing times.

As a result of this variation in mean processing time the predictive accuracy is highest when calibrating the model at a number of clients between the maximum and minimum mean processing times. This can be observed in figure 1 which shows the accuracy of predictions on the established server when calibrating the model at different numbers of clients. However the higher the number of clients used for the recalibration the more server capacity that must be taken offline to run the workload generators, application server and database server. In this case the maximum predictive accuracy when the model is recalibrated at a low overhead is at 125 clients. When the number of clients used for the calibration is reduced below 125 the predictive accuracy drops significantly. This is due to a discontinuity in the rate at which the mean response time increases with number of clients around the point at which max throughput is reached. The accuracy drops because the accuracy sample at this point is very inaccurate due to the point at which this discontinuity occurs being predicted incorrectly. Due to this discontinuity the predictive accuracy at the first maximum is slightly lower than that of the second maximum.

When the model is calibrated at 125 clients the mean processing times for the typical workload are measured as 4.675ms, 1.821ms and 0.638ms for the application server, database server and database server disk respectively. This results in a predictive accuracy of 84% on the new ($AppServ_S$) server architecture – see figure 2. A server capacity of 89 requests/second must be taken offline for this calibration for 2 minutes per service class. This is the equivalent of a Pentium III 450Mhz which is likely to be a low recalibration overhead for a modern resource management system.

4. The Historical Technique

The historical modelling technique involves sampling performance metrics (i.e. response times) and associating these measurements with variables representing the workload being processed and

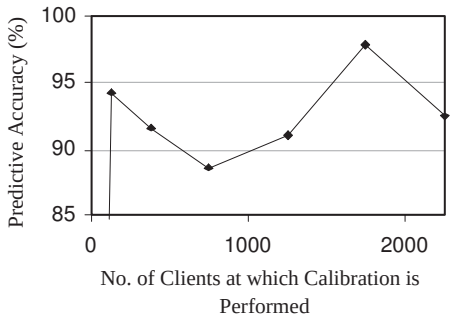


Figure 1. The predictive accuracy when recalibrating the layered queuing model at different loads

Server	CL (ms)	Lambda L (ms)
S	138.9	4E-06
F	84.1	0.0001
VF	10.7	0.0009

Table 1. Historical technique relationship parameters

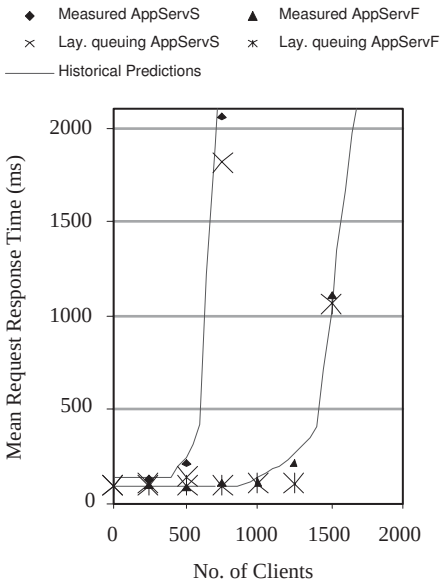


Figure 2. Performance predictions for new (AppServ S) and established (AppServ F) server architectures

the machines architecture. Historical models then define the relationships (i.e. linear/exponential equations) between the variables and metrics. In the case study the server workload variables are the number of clients and the percentage of buy requests, and the main server architecture variable is request processing speed. This results in three relationships.

Relationship 1 models the effect of the number of typical workload clients on the mean response time. It has been found that this relationship is best approximated using separate 'lower' and 'upper' equations for before and after max throughput:

$$mrt_L = c_L e^{(\lambda_L \times no\_of\_clients)} \tag{2}$$

$$mrt_U = \lambda_U \times no\_of\_clients + c_U \tag{3}$$

where mrt_L/mrt_U is the mean response time before/after max throughput respectively, and c_L, c_U, λ_L and λ_U are parameters that must be calibrated from historical data. The correct choice of the lower or the upper equation is made using a relationship between the number of clients and the servers throughput to calculate the number of clients at max throughput. It is also found that using a further breakdown of the possible system loads, so as to define a 'transition' relationship for phasing from the lower to the upper equation, can increase predictive accuracy as discussed in [5]. However the accuracy of such a relationship is not considered further here.

Relationship 2 models the effect of application server max throughput on relationship 1 as follows:

$$c_L = \Lambda(c_L) \times mx\_throughput + C(c_L) \tag{4}$$

$$\lambda_L = C(\Lambda_L) \times mx\_throughput^{\Lambda(\lambda_L)} \tag{5}$$

where $\Lambda(c_L)$, $C(c_L)$, $C(\lambda_L)$ and $\Lambda(\lambda_L)$ are parameters that must be calibrated from historical data. Parameters for the upper (linear) equations are calculated as follows. Given an increase/decrease in server max throughput of z%, λ_U is found to increase/decrease by roughly $1/z$%, and c_U is found to be roughly constant.

Relationship 3 models the effect of the percentage of buy requests in the workload on the servers max throughput. There is found to be a linear relationship between the percentage of buy requests on an established server and its max throughput. This is used to extrapolate $mx\_thr_E(b)$, the max throughput of an established server under a percentage of buy requests, b. The max throughput on a new server at a particular percentage of buy requests, $mx\_thr_N(b)$, is then calculated as follows, where a percentage of buy requests of 0 represents the typical workload:

$$mx\_thr_N(b) = \frac{mx\_thr_E(b)}{mx\_thr_E(0)} \times mx\_thr_N(0) \quad (6)$$

The remainder of this section investigates the calibration of historical models on a live system. This allows a historical model to be calibrated at a significantly smaller resource usage overhead than the layered queuing method as the only additional requirement on the system is to process the one (or more) response time sampling clients. The parameters in relationships 1 and 2 are calibrated by fitting least squares trend-lines to historical data from the established $AppServ_F$ and $AppServ_{VF}$ servers. The historical data consists of the max throughputs of each server and n_{udp}/n_{ldp} data points for the upper/lower equation of relationship 1 respectively. Each data point records the mean response time (averaged across n_s samples) of the typical workload at a numbers of clients.

The overall predictive accuracy is defined as the mean of the lower equation accuracy and the upper equation accuracy. It is found that accurate predictions can be made even when n_{udp} and n_{ldp} are both reduced to 2 and n_s is reduced to 50. The resulting parameters are shown in table 1. Figure 2 illustrates the mean response time predictions made using this calibration (including a transition exponential relationship for phasing between equations 2 and 3). A minimum of 100 samples per 'measured' data point are recorded. A good level of accuracy of 89% for the established servers and 83% for the new server is achieved. For these predictions the 50 samples for each data point were made sequentially (after a 1 minute warm-up period) using only one sampling client. This said, the calibration was completed in only 2 minutes. Relationship 3 can also be rapidly calibrated as this only requires one additional item of data; the max throughput of an established server under a heterogeneous workload. This is tested using LQNS predictions for historical data; specifically the max throughput of $AppServ_F$ under 25% buy requests (158 requests/second). The resulting predictive accuracy is 74% on the new server architecture.

5. Guidelines and Comparative Evaluation

It has been shown that both techniques can make accurate predictions at a low calibration overhead. Our guidelines for achieving these high levels of predictive accuracy using the layered queuing method are as follows based on the experimental analysis in section 3. The key variable is the number of clients at which the layered queuing model is calibrated. This is because the layered queuing technique assumes that the per-service class request processing times are constant at different server loads. However this was not found to be the case due to system (i.e. garbage collection) and thread overheads at small and large numbers of clients, respectively. As this variable, and hence the server load, is increased so does the calibration overhead. We have found that there tends to be a minimum number of clients that gives a high accuracy and low overhead (125 clients in our setup - see figure 1). The procedure in section 3 should be used to identify this point. Alternatively if spare machines

are available to calibrate the model the procedure can be used to locate the high overhead calibration point that gives the maximum accuracy (see figure 1). In our experimental setup this point was at 1750 clients.

Our guidelines for achieving accurate predictions at a low overhead using the historical method are as follows based on the experimental analysis in section 4. It is necessary to sample the response times of established servers for both before and after max throughput is reached due to the discontinuity in the response time scalability graph at this point (see figure 2). It is also necessary to use two (or more) established application server architectures for calibration so as to be able to extrapolate a trend-line (unlike the layered queuing technique which only requires one). Further, when sampling the response times of an established server for either before or after max throughput it is necessary to include samples at both low and high numbers of clients so as to get a sufficient spread of data points from which to draw a trend-line. It has also been found that the predictive accuracy before max throughput is reached tends to be less than the predictive accuracy after max throughput is reached. This is due to the predictions being made using exponential and linear relationships respectively. Initial experiments have shown that exponential predictive accuracy increases to linear predictive accuracy levels if three or more (no. of clients, mean response time) data points are used for calibration as opposed to the current two. We therefore recommend that a minimum of two/three data points (with at least 50 samples per data point) be used when calibrating linear/exponential trend-lines, although in practice the more data points used the better.

When selecting which performance prediction technique to use we recommend that the following criteria be considered: recalibration requirements; the responsiveness of predictions; the systems which can be modelled, the metrics which can be predicted, the ease of use of each technique and the performance modelling expertise required. Model recalibration has been considered above and in the previous two sections; the following is a summary of how the techniques differ in the other categories. For a more detailed discussion the reader is referred to [4].

There are a number of functional limitations with the layered queuing technique that should be taken into account when selecting a technique. It requires significant CPU time to make the mean response time predictions (up to 3 seconds on an Athlon 1.4Ghz under a convergence criterion of 20ms in these experiments), whereas the historical predictions are almost instantaneous. It is also more difficult to model applications that cache significant amount of database data at the application server (as opposed to applications such as the Trade benchmark which access the majority of database data directly so as to avoid data inconsistencies if the application server crashes). This is because the number of calls to the database must be a constant in the layered queuing model, whereas if a cache is used this value will depend on the cache miss rate. Using the historical method the size of the application servers cache can be recorded as an extra variable. Relationships can then be added to approximate the historical relationship between the performance metrics, this new variable and the existing variables, using the techniques presented in section 4. Another limitation of the layered queuing technique is that the important class of percentile response time metrics cannot be predicted directly. In contrast the historical technique can extrapolate from and hence make predictions for a wide range of metrics. However layered queuing models are also significantly easier to create with a minimum level of performance modelling expertise than a historical model. This is because creating a historical model involves specifying and validating how predictions will be made, whereas once a system's queuing network configuration is specified layered queuing models can be solved automatically. The layered queuing technique may therefore be preferable when there is a shortage of either time or performance modelling expertise when creating the model.

6. Conclusion

This paper comparatively evaluates the layered queuing and historical techniques for predicting response times of dynamic enterprise information systems on new server architectures. Based on detailed experimental analysis we provide guidelines for selecting a technique and obtaining low overhead high accuracy predictions. The combination of an established system model, a popular middleware (IBM Websphere) and a distributed enterprise benchmark based on best practices (the Websphere Performance Benchmark Sample) should make this work of relevance to a wide range of enterprise information systems. This is also, to the best of our knowledge, the only quantitative comparison of these two classes of prediction technique on this benchmark. Future work includes evaluating the strengths and weaknesses identified with each technique on different types of prediction-enhanced workload management algorithm.

References

[1] J. Aman, C. Eilert, D. Emmes, P. Yocom, D. Dillenberger: Adaptive Algorithms for Managing a Distributed Data Processing Workload. IBM Systems Journal. 36(2), 1997, 242-283.

[2] Z. Liu, M.S. Squillante, J. Wolf, On Maximizing Service-Level-Agreement Profits, Proc. ACM Conference on Electronic Commerce (EC01), Florida, USA, October 2001

[3] M. Goldszmidt, D. Palma, B. Sabata, On the Quantification of e-Business Capacity, Proc. ACM Conference on Electronic Commerce (EC01), Florida, USA, October 2001

[4] D.A. Bacigalupo, S.A. Jarvis, L. He, D.P. Spooner, D.N. Dillenberger, G.R. Nudd, An Investigation into the Application of Different Performance Prediction Methods to Distributed Enterprise Applications, The Journal of Supercomputing, 34:93-111, 2005

[5] D.A. Bacigalupo, S.A. Jarvis, L. He, G.R. Nudd, An Investigation into the Application of Different Performance Prediction Techniques to e-Commerce Applications, Workshop on Performance Modelling, Evaluation and Optimization of Parallel and Distributed Systems, Proc. 18th IEEE Int. Parallel and Distributed Processing Symposium (IPDPS04), New Mexico, USA, April 2004

[6] J.D. Turner, D.A. Bacigalupo, S.A. Jarvis, D.N. Dillenberger, G.R. Nudd, Application Response Measurement of Distributed Web Services, Int. J. of Computer Resource Measurement, 108, 2002, 45-55

[7] D. Menasce, Two-Level Iterative Queuing Modeling of Software Contention, Proc. 10th IEEE Int. Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS02), Texas, USA, October 2002

[8] Y. Diao, J. Hellerstein, S. Parekh, Stochastic Modeling of Lotus Notes with a Queueing Model, Proc. Computer Measurement Group Int. Conference (CMG01), California, USA, December 2001

[9] C.M. Woodside, J.E. Neilson, D.C. Petriu, S. Majumdar, The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-like Distributed Software, IEEE Trans. On Computer, 44(1), 1995, 20-34

[10] F. Sheikh, M. Woodside, Layered Analytic Performance Modelling of a Distributed Database System, Proc. Int. Conference on Distributed Computing Systems (ICDCS97), Maryland USA, May 1997

[11] T. Liu, S. Kumaran, J. Chung, Performance Modeling of EJBs, Proc. 7th World Multiconference on Systemics, Cybernetics and Informatics (SCI03), Florida USA, 2003

[12] M. Endrel, IBM WebSphere V4.0 Advanced Edition Handbook, IBM Int. Technical Support Organisation Pub., 2002. Available at: http://www.redbooks.ibm.com/

[13] IBM Websphere Performance Sample: Trade. Available at www.ibm.com/software/info/websphere/

[14] L. Zhang, C. Xia, M. Squillante, W. Nathaniel Mills III, Workload Service Requirements Analysis: A Queueing Network Optimization Approach, Proc. 10th IEEE Int. Symposium on Modeling, Analysis, & Simulation of Computer & Telecommunications Systems, Texas, USA, October 2002

Scheduling for Heterogeneous Networks of Computers with Persistent Fluctuation of Load

R. Higgins[a*] A. Lastovetsky[a]

[a]School of Computer Science and Informatics, University College Dublin, Dublin 4, Ireland

In this paper we present a model of performance for nodes in a heterogeneous Network of Computers (NOC). Unlike a dedicated cluster a NOC is made up of machines that have varying levels of integration with the rest of a general purpose network. This integration results in different load fluctuations on nodes in the NOC. Our model aims to represent how these routine fluctuations effect performance and we demonstrate the construction of the model and its use in the design and implementation of a parallel applications.

1. Introduction

Networks of Computers (NOCs) provide high performance parallel computing capabilities without the significant investment of acquiring a dedicated cluster. They are typically built from a wide variety of machines existing in a campus, office or some other general purpose network. This heterogeneity between the computers exists at a number of levels as a NOC can be built using any computing resource available.

Performance models for heterogeneous NOCs attempt to represent the differences between the speeds of machines in the network and describe topology of the network interconnect. In this paper we are concentrating on modelling the speed of a non-dedicated machine. Our model describes how the processing speed may vary during the execution of some problem. We present a method of partitioning a data-parallel application which attempts to balance the load on heterogeneous processors in the presence of load fluctuations.

Traditional heterogeneous parallel programming systems estimate performance of individual nodes in the computing network using a single benchmark number. This number is calculated by timing the execution of some critical section of an algorithm[2] or some standard test code[1]. Workload is distributed proportionally according to the benchmarks of the machines involved in the computation. The traditional model shows some weakness on machines that have a high level of integration with the network. A NOC may consist of a number of non-dedicated resources that have some role in the wider network: they may be acting as a network file server, a users personal desktop, mailserver, etc. These machines experience fluctuations in their workload operating in their different roles. The single benchmark model of performance must be built in a short period of time. On a machine operating under a constant fluctuation in load this benchmark may be run during a period of higher or lower than average load. The conditions that an actual computation executes under are quite different to those that a benchmark would. An actual computation executes for a much longer period of time. The fluctuating load that it must contend with will average out over this period to a more steady level. We aim to improve performance of jobs run on a NOC by using a more detailed model of processor performance that represents the varying speed of a machine and by partitioning a computation in a manner that accounts for possible changes in external loads across the NOC.

*This research is funded jointly by the Irish Research Council for Science, Engineering & Technology and IBM's Center for Advanced Studies in Dublin.

We are building upon concepts introduced in [4]. Our model represents the effect of fluctuations in workload on the performance of a machine in a NOC. In place of the single benchmark we use a more detailed performance function as presented in [3]. The performance function describes the execution speed of an application on a machine as the size of the data operated on is increased. We adjust it to create a band of performance that describes a range of possible execution speeds as the problem size increases. Machines that experience large fluctuations in workload will have their performance represented by a wider band than those that experience a more consistent level of workload. The bands are used to find a partitioning of a problem that is optimal for the widest range of load fluctuations across all machines in the network.

The remainder of this paper is organised as follows. In section 2 we describe the procedure to build the band model using load history and a performance function. A algorithm to solve the problem of partitioning with the band model is detailed in section 3. Section 4 presents analysis on the performance increase attainable using the band model. Conclusions and direction for future work are offered in Section 5.

2. Building The Model

The construction of the performance band model requires a number of steps. It is built from two components: a performance function and a pair of load functions. The performance function consists of a number of experimentally obtained benchmarks (execution times) for problems of increasing size. The load functions represent the maximum and minimum average load experienced by a machine over an increasing period of time. For each point in the performance function, using the load functions, we find the maximum and minimum load it might encounter during the problem's time executing. These loads are used to adjust the performance function for worst case performance, contending with a high level of load, and best case performance, contending with a low level of load. In the following subsections we provide more detail on the constituents of the band model, how they are obtained and how they are combined.

2.1. Performance Functions

The first step in the creation of the performance band for a machine is to build a piecewise linear function of optimal execution speed with respect to problem size. The optimal execution speed $s_o(x)$ is the rate at which a problem of size x is solved on an idle processor. The function requires a number of experiments to measure the optimal running time $t_o(x)$ of the application as x increases. $t_o(x)$ can be measured in any UNIX or UNIX-like environment using the **time** utility or the **getrusage()** system call. These functions return statistics maintained by the operating system kernel on the resources used by an application including the time a process spent actively executing on the CPU. This is equivalent to its optimal running time, $t_o(x)$. $s_o(x)$ is equal to the volume of computations for a problem size x divided by $t_o(x)$. We will not consider the construction of $s_o(x)$ any further in this paper, taking it as already provided. Choosing an optimal set of problem sizes with which to measure the execution speed is the subject of a paper pending publication. figure4a.eps

The performance band is built from $s_o(x)$ by adjusting each experimentally obtained point for two scenarios: worst and best predicted performance. Our prediction of performance is based on a history of observed load averages. Using the observations we build two load functions: $l_{max}(t)$ and $l_{min}(t)$. These functions represent the maximum and minimum average load over a period of t time units. We use these functions to find the average loads that would have occurred during the execution of a problem of size x. The maximum and minimum averages are used as factors to adjust $s_o(x)$, resulting in two functions that define the limits of the performance band: $s_{max}(x)$ and $s_{min}(x)$.

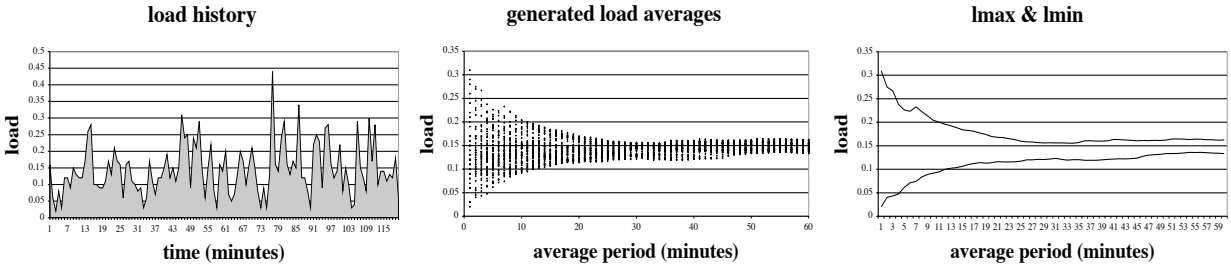


Figure 1. Steps in creation of load functions: first collect a history of load observations, then calculate all load averages of increasing periods, finally extract maximum and minimum piece-wise linear functions: l_{min} and l_{max}.

2.2. Load Functions

The load average on a UNIX or UNIX-like machine is described as: 'the number of processes in the system run queue averaged over various periods of time'.[2] The kernel of the operating system maintains averages with time periods of one, five and fifteen minutes. The averages are exponentially-damped, sampling the run queue length at an OS dependent frequency. They are available through system utilities such as **uptime** and library calls such as **getloadavg()**. A history of load fluctuations is kept by sampling the one minute load average every δ time units. A load average of a time period t can be calculated from the observations by averaging a sequence of $\frac{t}{\delta}$ observations.

$l_{max}(t)$ and $l_{min}(t)$ represent load averages of increasing period t up to some limit w. This limit may be defined as the running time of the largest problem permitted to run on the machine that the functions represent. The size of the history of load averages observations used is defined by h. A sliding window of length w is passed over the h observations and at each position of the window a set of load averages, with periods from δ increasing to w are calculated. A one minute average would be given by the first load observation in the window, the two minute average would be calculated from the average of the first and second observation, and so on. As the window moves over h observations it creates load averages with periods of $\delta, 2 \times \delta, ..., w$ until it begins to slide over the edge of the recorded history, at which point the range of load averages calculated decreases. From these calculated averages a maximum and minimum average for each time period $\delta, 2 \times \delta, ..., w$ are extracted and these values are used to build the piecewise linear functions $l_{max}(t)$ and $l_{min}(t)$.

More formally, if we have a sequence of observed loads $l_1, l_2, ..., l_h$, then a matrix A of calculated load averages is defined as follows:

$$A = \begin{pmatrix} a_{1,1} & . & . & a_{1,h} \\ . & . & . & \\ . & . & & \\ a_{w,1} & & & \end{pmatrix} \text{ where } a_{i,j} = \frac{\sum_{k=j}^{i+j-1} l_k}{i \times \delta} \text{ for all } i = 1...h; j = 1...w \text{ and } i + j \leq h + 1$$

$l_{max}(t)$ and $l_{min}(t)$ are then defined by the maximum and minimum calculated loads from a row t in the matrix A. Points are connected in sequence to give a continuous piecewise linear function. The tth period load averages are given by:

$$l_{max}(t) = \max_{i=1}^{h}(A_{i,t})$$

$$l_{min}(t) = \min_{i=1}^{h}(A_{i,t})$$

[2] From the 'getloadavg' man page entry, section 3 (library calls), Linux Programmers Manual.

The load statistic is maintained by the operating system continuously, there is no additional computation involved our measurement of the load at each δ interval. In comparison to the cost of the initial generation of the performance function, computing the load functions and the resulting performance band is trivial. The benefit of the performance band can be had for almost not cost at all.

2.3. Performance Bands

Given the functions relating to optimum speed of execution: $s_o(x)$ and $t_o(x)$ and the load fluctuation functions $l_{max}(t)$ and $l_{min}(t)$, we now wish to calculate the performance band. An application will execute for a certain period of time, the minimum being $t_o(x)$ when there is no external load on the executing machine. As load increases the executing time also increases. We need to find the points at which the executing time under a given load matches the predicted maximum or minimum load for that time period. The executing time of an application $t_e(x)$ can be calculated by dividing $t_o(x)$ by a measure of the CPU availability. We may estimate CPU availability for a single threaded application based on a load average with the equation (1), where n is the number of processors on a machine. This estimate is not perfect as it's simplicity does not reflect the complex job a kernel does in scheduling processes. In most cases however, the conversion from load average to CPU availability is accurate enough for the purposes of problem partitioning [7].

$$a(l) = \begin{cases} 1 & (l \geq n-1) \\ \frac{n}{1+l} & (l > n-1) \end{cases} \tag{1}$$

Using the availability, we may plot a function of execution time: $t_e(l) = \frac{t_o(x)}{a(l)}$ for a particular problem size x. The points where this line intersects the load fluctuation functions $l_{max}(t)$ and $l_{min}(t)$ correspond to the factors by which we must adjust the optimal speed of execution to create the performance band (Figure 2).

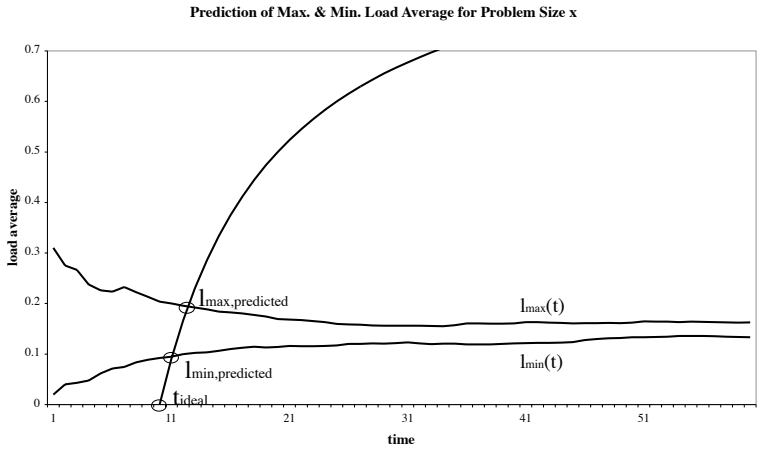


Figure 2. Finding the average load that effects an application over the duration of it's execution.

Using these points, $l_{max,predicted}$ and $l_{min,predicted}$, we adjust the optimal speed to give us the upper and lower limits of the performance band:

$$s_{max}(x) = s_o(x) \times a(l_{min,predicted}) \tag{2}$$

$$s_{min}(x) = s_o(x) \times a(l_{max,predicted}) \tag{3}$$

3. Using The Model

In [3] the partitioning of a problem using a functional performance model was introduced. It was demonstrated that the proportional partitioning of a problem occurred when a line through the origin intersected the performance functions of each machine at points corresponding to their assigned workload (figure 3). Performance bands may be considered as a sets of performance functions occurring between between the maximum and minimum performance levels $s_{max}(x)$ and $s_{min}(x)$. Given a particular distribution of work there may be many combinations performance levels across machines in the NOC that result in a perfectly proportional distribution. Figure 4 illustrates two combinations of performance levels within a band, for which some distribution is optimal. The aim in partitioning a problem with the band model is to find the distribution that maximises the amount of possible combinations of performance where the distribution will remain optimal. These performance levels correspond to levels of workload created on the a machine due to it's non-dedicated status. The distribution allows the maximum amount of fluctuation in load without harming the balance of the distribution.

Calculating the Distribution Arc As can be seen in figure 4 there is an common arc α between which the distribution of work remains proportional. In finding the best possible distribution we wish to maximize the size of this arc. For the two processor example in figure 5, the angle of the arc is determined by lines drawn through the origin to points on the bands. These points given by the work assigned to the respective processors. The arc is always defined by the shallowest line which intersects a maximum speed function, subtracted from the steepest line which intersects a minimum speed function (4).

$$\alpha = \min\left(\arctan\left(\frac{s_{max,0}(w_0)}{w_0}\right), \arctan\left(\frac{s_{max,1}(w_1)}{w_1}\right)\right) - \max\left(\arctan\left(\frac{s_{min,0}(w_0)}{w_0}\right), \arctan\left(\frac{s_{min,1}(w_1)}{w_1}\right)\right) \quad (4)$$

Maximising this equation via differential analysis was considered too complex so a heuristic algorithm was used to find the optimal solution. Observation of the problem space has shown no occurrence of local maxima close to the global maximum. Assuming this to be true, a standard hill climbing algorithm [5,6] was chosen to search for this maximum. Experiments with the algorithm have not conflicted with the assumption, though further work would be required to prove it true for all situations. To speed the search for a maximum α the algorithm is guided by initially distributing the workload according to the the average speed of each workstation: a midpoint between $s_{max}(x)$ and $s_{min}(x)$ at an arbitrary value of x. From this point the hill climbing algorithm quickly finds the optimal distribution.

4. Analysis

In our analysis we compare the theoretical execution time of a job partitioned using a single benchmark with a job partitioned using our band model. We suppose that the single benchmark is taken directly before the execution of the parallel job. Such benchmarks are short and subject to load fluctuations. Given the load fluctuation functions of a machine we calculate a set of benchmarks, ranging from ones that may be measured during a high load to a low load. Combinations of these benchmarks are used to partition the job and a running time is calculated. The load combinations in the figures presented are shown on the x and y axes. The speed up gained by using the band model instead of the load-effected benchmarks is shown on the z axis.

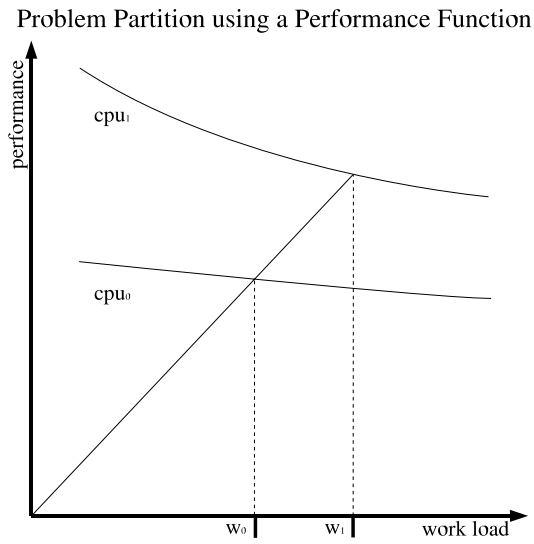


Figure 3. Proportional problem partitioning with performance functions.

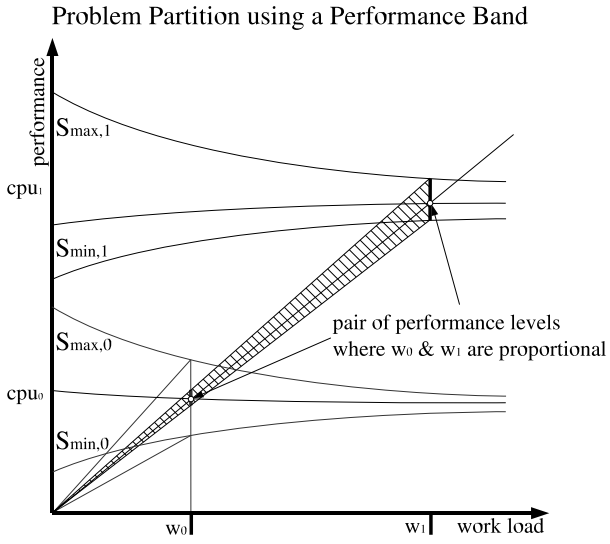


Figure 4. Two pairs of performance levels for which distribution is proportional.

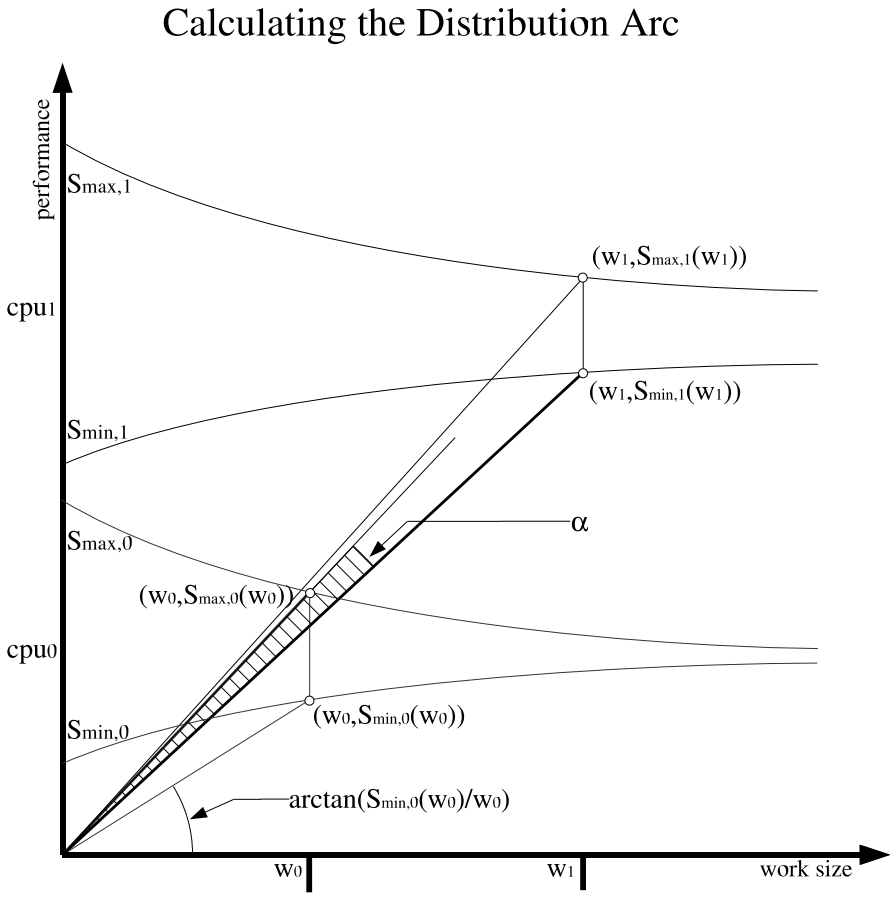


Figure 5. Calculating the size of the common arc α for a data distribution between two processors.

The comparison of the single benchmark and band model is done for three scenarios. A job is partitioned between two machines where: both machines are experiencing: a low level of load fluctuation (figure 6), a high level of load fluctuation (figure 7) and a combination of low and high level of load fluctuation (figure 8). The machines themselves are identical. This homogeneity exists so that we may focus on the effects of heterogeneity in the workloads on the machines. From these plots we can see the conditions under which the band model out performs the benchmark, namely when higher levels of load fluctuation exist on the NOC. We found that for scenarios where both machines have similar levels of load fluctuation, the band model is does not offer very much benefit over a single benchmark. An average 1% speedup was calculated. As the variance between the load fluctuations increases the speed up increases. With two busy machines an average 7% speedup was calculated and 9%, ranging as high as 35%, when an idle machine was paired with a busy machine.

5. Conclusions

In these paper we have presented a method of representing load fluctuation using a performance band. We have demonstrated the construction of the performance band and using load observations and suggested a measure for the optimality of a distribution using this performance band: that of the widest angle α.

Our analysis has shown that a scheduling created by maximizing α outperforms one created using single benchmarks under circumstances where load fluctuations on some of the machines in a NOC are high. When the loads on all machines participating in the computation are relatively steady the performance gains are unclear. In such situations, using the band model only adds complexity to the scheduling. A hybrid model of performance could identify situations where the band is unnecessary and use simpler methods in those circumstances.

From this point there are a number of directions our research may take. The measure of optimality α currently only considers the common arc between all processors. This may be adjusted to also maximise arcs that overlap between a subset of the total number of processors. Doing so would add value to distributions that allow fluctuation in parts of the NOC, but not all. As the number of processors in the NOC is increased, finding a distribution that allows a large load fluctuation on all processors is unrealistic, so this kind of measure may be required.

The load functions $l_{max}(t)$ and $l_{min}(t)$ are built from the extremes of observed loads, however between these extremes there exists a probability curve where load is more likely to occur. Currently the optimality measure gives equal importance to all parts of the band. It is probably more beneficial to impart greater value on distributions that allow fluctuation regions of the band where load is more likely to occur than where it is less likely, at the extremes.

Finally, for more complex measures of goodness the problem space may not suit the simple hill climbing method of optimising the distribution. More elaborate Evolutionary Algorithms should examined and implemented to efficiently maximise goodness factor.

References

[1] H. Casanova and J. Dongarra. NetSolve: A network server for solving computational science problems. Technical Report CS-96-328, Knoxville, TN 37996, USA, 1996.

[2] A. Lastovetsky. Adaptive parallel computing on heterogeneous networks with mpc. *Parallel Comput.*, 28(10):1369–1407, 2002.

[3] A. Lastovetsky and R. Reddy. Data partitioning with a realistic performance model of networks of

heterogeneous computers. In *Proceedings. 18th International Parallel and Distributed Processing Symposium*, page 104, 2004.

[4] A. Lastovetsky and J. Twamley. Towards a realistic performance model for netowrks of heterogeneous computers. In *International Symposium on High Performance Computational Science and Engineering*, 2004.

[5] N. J. Nilsson. *Problem-Solving Methods in Artificial Intelligence*. McGraw-Hill, New Yourk, 1971.

[6] E. Rich and K. Knight. *Artificial Intelligence*. McGraw-Hill, New Yourk, second edition, 1991.

[7] R. Wolski, N. T. Spring, and J. Hayes. Predicting the CPU availability of time-shared unix systems on the computational grid. *Cluster Computing*, 3(4):293–301, 2000.

Appendix

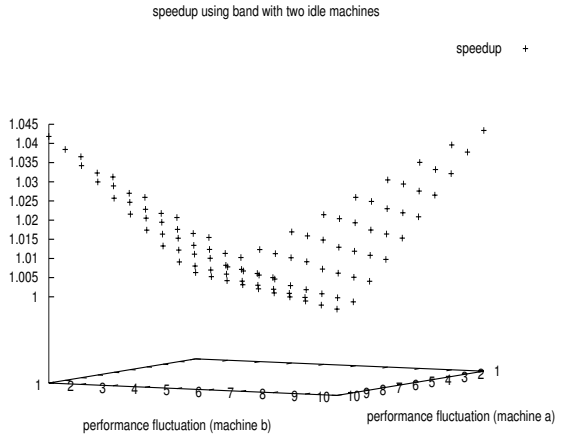


Figure 6. Speed up using two idle processors (1% average).

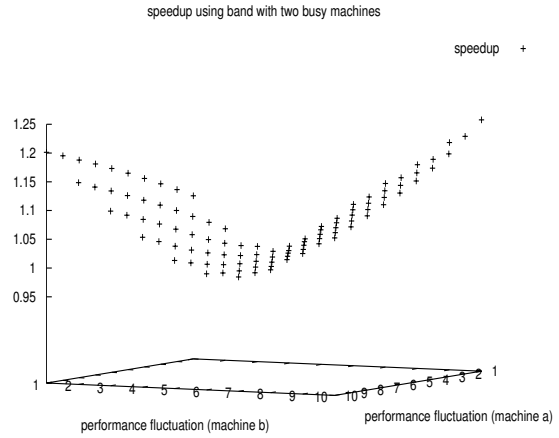


Figure 7. Speed up using two busy processors (7% average)

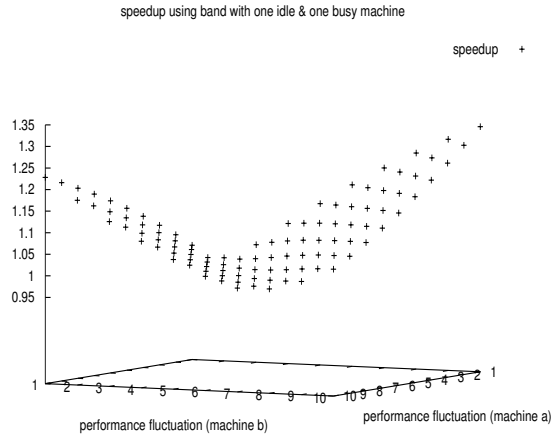


Figure 8. Speed up using one busy and one idle processor (8% average)

Analysis and Optimization of Yee_Bench Using Hardware Performance Counters

Ulf Andersson[a], Philip Mucci[a,b]

[a]PDC, KTH, SE-100 44 Stockholm, Sweden, ulfa@nada.kth.se

[b]Innovative Computing Laboratory, University of Tennessee, mucci@cs.utk.edu

In this paper, we report on our analysis and optimization of a serial Fortran 90 benchmark called Yee_bench. This benchmark has been run on a variety of architectures and its performance is reasonably well understood. However, on AMD Opteron based machines, we found unexpected dips in the delivered MFLOPS of the code for a seemingly random set of problem sizes. Through the use of the Opteron's on-chip hardware performance counters and PapiEx, a PAPI based tool, we discovered that these drops were directly related to high L1 cache miss rates. The high miss rates could be attributed to the fact that in the two compute kernels of the application there are references to three equal-sized dynamically allocated arrays which compete for the same set in the Opteron's 2-way set-associative cache. We validated this conclusion by accurately predicting those problem sizes that exhibit this problem. Furthermore, we were able to alleviate these performance anomalies using variable intra-array padding to effectively accomplish inter-array padding. We conclude with some comments on the general applicability of this method as well how one might improve the implementation of the Fortran 90 ALLOCATE intrinsic to handle this and other cases of set-associativity conflicts.

1. Introduction

In this paper, we report on the analysis and optimization of Yee_bench [1,2] using a Performance Application Programming Interface (PAPI) [7] based tool called PapiEx (PAPI Execute) [8]. We show how we tracked down a number of performance dips when Yee_bench was run on an AMD Opteron and the problem size was varied.

Yee_bench is a benchmark developed at the Center for Parallel Computers (PDC) in Stockholm, Sweden. It implements the core of the Finite-Difference Time-Domain (FDTD) method [5,6,9] for the Maxwell equations, commonly used in computational electromagnetics. See Appendix A for a listing of the core of Yee_bench. The Yee_bench code is strongly bound by memory bandwidth and it's results show a strong correspondence [1] with the numbers produced from the STREAM2 microbenchmark [4]. Yee_bench is widely used at PDC as part of the architectural evaluation process when purchasing new hardware as many of our applications are bound by memory bandwidth.

To achieve the best possible speed-up and scale-up for the parallel version of Yee_bench, it is crucial to understand the performance of the serial code and how it depends on the problem size. If this behavior can be understood, we will know when to use padding to avoid unsuitable local problem sizes in the parallel code. Many FDTD applications strives to use as much memory as possible, hence scale-up is a relevant measure of parallel performance. Thus it follows that behavior for large problem sizes are more important than performance for small problem sizes.

These dips in performance occurred with no apparent regularity and thus we looked to hardware performance monitors (through the use of PAPI and PapiEx) to provide us with additional insight. PapiEx is a portable and easy-to-use command line tool to monitor the performance counters for

any executable. No source code instrumentation is needed although a simple instrumentation API is provided. Through the information gathered with `PapiEx`, we were able to characterize and subsequently alleviate these performance anomalies in `Yee_bench`.

We also present the performance of `Yee_bench` on the Intel Itanium-2 and the Intel Xeon EM64T for completeness.

2. AMD Opteron

2.1. Technical data

For our AMD runs we used one CPU of a four-way Opteron Processor 846 running at 2.0 GHz. The memory subsystem consists of a two-way set-associative 64 kbytes L1 cache, a sixteen-way set-associative 1 Mbytes L2 cache, and 8 Gbytes of DDR 333 with 2 Gbytes per CPU. The L1 cache line length was 64 bytes.

The compiler used was version 5.2-4 of `pgf90`. The following options were used: `-fast -fastsse -tp=amd`. Similar behavior was observed with the `pathf90` compiler on other Opteron systems. The OS used was SUSE 9.1 with Linux kernel 2.6.8.

2.2. Results

The upper part of Figure 1 displays the 64-bit precision performance of the original (no padding) version of `Yee_bench` [1]. The computational domain used is a cube and thus $N_x = N_y = N_z \equiv N$.

The results for the original code in Figure 1 are representative of all Opteron results achieved for `Yee_bench`. Based on our previous experience on other architectures, we expect poor performance whenever N or $N + 1$ is a power-of-two if no padding is used [1]. On some architectures we get poor performance whenever N or $N + 1$ contains at least four factors of 2 due to low cache hit rate (see [1]). This phenomena is well understood and easy to avoid using padding. However, in Figure 1 we have poor performance for many more problem sizes than previously encountered.

2.3. L1 Cache hit rate

We initially suspected intra-array way conflicts in the L1 cache. So we ran a different version of `Yee_bench`, one that contained intra-array padding. Each of the three electric field arrays were padded identically, as were the three magnetic field arrays. We also tried the stock version of the code, except that we introduced compiler directives to do the padding. Neither had any appreciable affect on performance, which ruled out intra-array conflicts. However, such tremendous drops in performance could only be caused by L1 misses. We therefore decided to measure the L1 cache hit rate for all problem sizes. The result is displayed in the lower part of Figure 1. The L1 cache hit rate was computed from `PapiEx` output as:

$$\frac{\text{PAPI\_L1\_DCH}}{\text{PAPI\_L1\_DCH} + \text{PAPI\_L1\_DCM}}. \tag{1}$$

There is clearly a correspondence between performance (GFLOPS) and L1 cache hit rate. The question now becomes: Why do we get so low L1 cache hit rate for certain problem sizes?

2.4. L1 cache hit rate analysis

We will first determine what is the best possible L1 cache hit rate for the loops (listed in Appendix A) in the kernel of `Yee_bench`. Each iteration of these triple nested loops contains three stores and ten loads. Four of the ten loads have already been used in earlier iterations, while six of the ten loads are new. Actually, there appear to be two more “loads”, but these values where used in

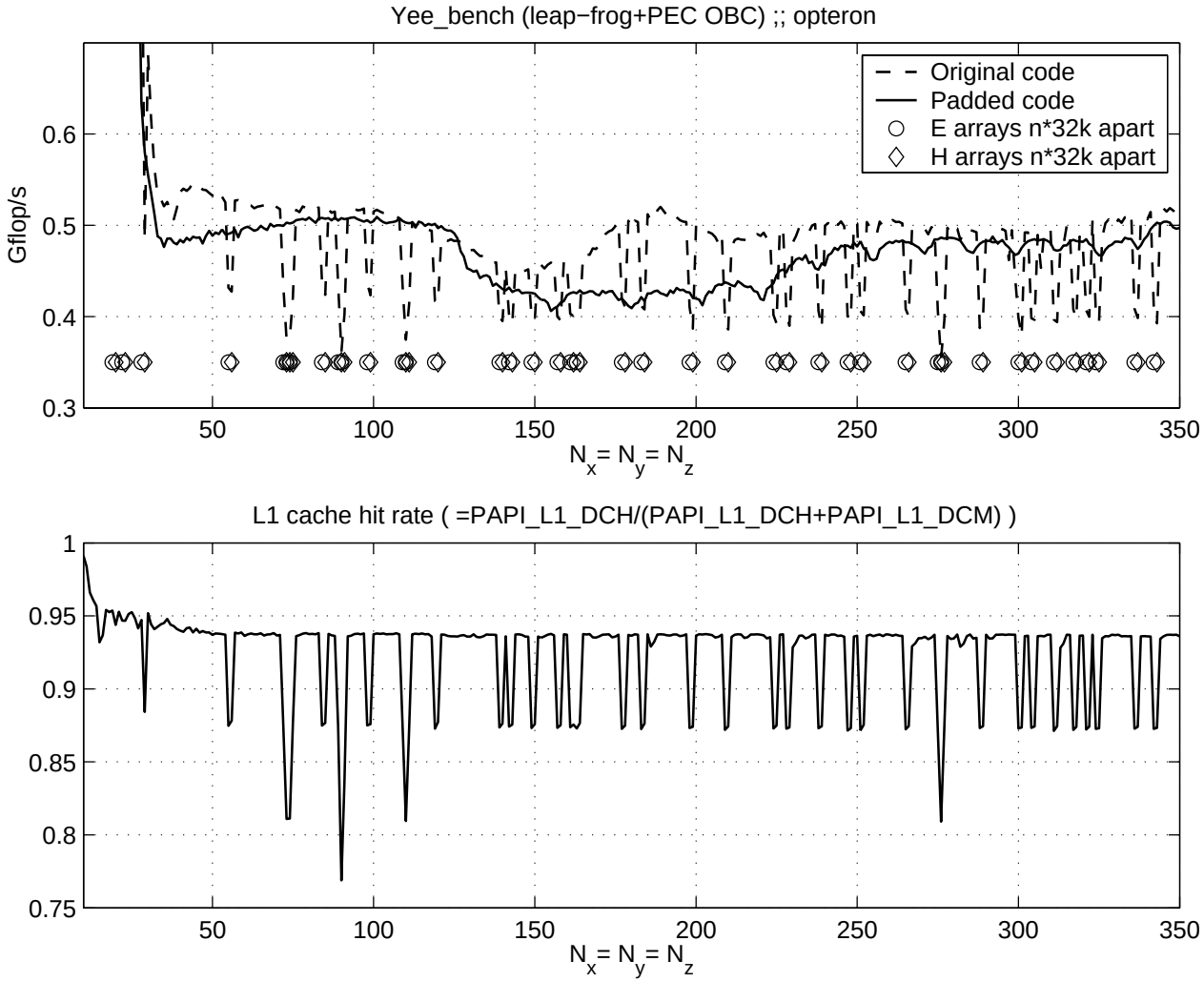


Figure 1. `Yee_bench` performance and L1 cache hit rate on the AMD Opteron.

the preceding iteration and can therefore be expected to reside in registers. This conjecture has been verified on several computers using hardware performance counters.

Each L1 cache line contains eight 64-bit precision floating-point values. This means that one of eight loads is a compulsory cache miss for each of the six new values. The four loads of already used values will always, in the ideal case, already be in cache. We expect every store to be a cache hit. This gives us a load hit rate of:

$$\frac{7}{8}\frac{6}{10} + \frac{4}{10} = \frac{37}{40} \approx 92.5\%\,, \tag{2}$$

and a total L1 cache hit rate of:

$$\frac{10}{13}\frac{37}{40} + \frac{3}{13} \approx 94.2\%\,. \tag{3}$$

The measured L1 cache hit rate for the good cases in Figure 1 is around 93.7 %, which is pretty close to our estimate in (3), which is an upper limit. It is not reasonable to expect the four loads of already used values to always be cache hits due to capacity limitations. Two of these values was used

in the preceding iteration for the middle loop and two values were used in the preceding iteration for the outer loop. Hence the may have been evicted from cache.

If none of the ten loads give a hit when we load the first element of a cache line we get a load hit rate of 7/8 and a total hit rate of:

$$\frac{10}{13}\frac{7}{8} + \frac{3}{13} \approx 90.4\% \tag{4}$$

The low cache hit rate values in Figure 1 are below the estimate in (4) which indicates that we have cache line contention. In the following section we will explore whether it is possible to predict for which values of N this will happen.

2.5. Allocation analysis

By using the utility function `LOC()`, which returns the address of a data item, we established that the electromagnetic arrays are always allocated with a distance that is a multiple of the pagesize (4 kbytes) when the arrays are larger than 128 kbytes. Considering that the L1 cache is two-way set-associative, we notice that if the three cache lines containing `Ex(i,j,k)`, `Ey(i,j,k)` and `Ez(i,j,k)` belong to the same set, we have contention for the same cache line.

The L1 cache is 64 kbytes. If we divide this by two due to the set-associativity, we get 32 kbytes or eight pages. Hence we postulate that we will get poor performance if the distance between the different arrays is a multiple of 32 kbytes. In Figure 1, we have indicated for which problem sizes we expect this to happen. To be able to do this prediction, it is important to note that there is a bit of overhead on each allocation. We have measured this overhead to be 80 bytes.

2.6. Padding

The original version of `Yee_bench` allowed for internal padding of the electric and magnetic field arrays. However, it used the same padding on all three electric field arrays and similarly for the three magnetic field arrays. This padding procedure was used in order to avoid having leading dimension that were powers-of-two. Due to cache-line contention, it was necessary to find a way to make sure that the iterations over the three dimensional arrays didn't start at the same place. However, the `ALLOCATE` statement in both the Portland Group compiler and the Pathscale compiler always returned data on a page boundary. In order to reduce the amount of changes to the code, we decided to use the existing padding infrastructure but pad each array differently. By doing so, we could guarantee that each iteration, except the first iteration, computed at different offsets into each array. With these changes, contention should only occur during the first iteration of the inner loop (see Appendix A) and the padding would handle subsequent iterations.

To achieve the individual padding we used this allocation scheme:

```
Hx(1:nx  +padHx(1),1:ny  +padHx(2),1:nz  +padHx(3))
Hy(1:nx  +padHy(1),1:ny  +padHy(2),1:nz  +padHy(3))
Hz(1:nx  +padHz(1),1:ny  +padHz(2),1:nz  +padHz(3))
Ex(1:nx+1+padEx(1),1:ny+1+padEx(2),1:nz+1+padEx(3))
Ey(1:nx+1+padEy(1),1:ny+1+padEy(2),1:nz+1+padEy(3))
Ez(1:nx+1+padEz(1),1:ny+1+padEz(2),1:nz+1+padEz(3))
```

Figure 1 displays the results for `padEx=padHx=(/1,0,0/)`, `padEy=padHy=(/2,0,0/)`, and `padEz=padHz=(/3,0,0/)`. We see that all the large dips have disappeared. However, for $150 < N < 250$ we see a loss of performance compared to the good cases of the original code. For the most important problem sizes, the large ones, we see a considerable improvement in performance

for the padded code version. Since we are able to predict when padding is needed we can choose to use it only when it is needed.

Here we have effectively achieved inter-array padding by doing intra-array padding. While complex, this technique is reasonably portable and could be implemented easily by the compiler during it's loop optimization phase.

2.7. Building a Better Fortran 90 ALLOCATE

At the time of this work, neither the Portland Group compiler nor the Pathscale compiler provided any mechanisms to do inter-array padding or otherwise change the behavior of the allocator. In theory, a compiler could be modified to provide hints to the allocator based on possible conflicts detected during the loop nest optimization phase. However, this would require significant complexity since the allocator could easily be in another module, requiring the modifications to be deferred until a whole program optimization phase (or global intraprocedural analysis phase). Additionally, this could cause significant overhead as a program could contain hundreds if not thousands of allocates and loop nests, each of which would have to be dealt with. Instead of making significant modifications to the complier, it is a better choice to simply waste some memory and return locations offset by one or more cache lines. Algorithm 2.1 implements such a scheme by maintaining a static local variable that contains the number of lines in one page. For multithreaded programs, we need not be concerned with atomically updating the "pad" variable, as this would result in only the occasional allocation with the same padding. This allocator need not be called for smaller allocations, say, under one page.

Algorithm 2.1: ALLOCATE($bytes$)

static $pad \leftarrow 0$
local $total\_bytes, num\_pages\_padded, leftover, mem, address$

$total\_bytes \leftarrow bytes + pad * L1LINESIZE$
$num\_pages\_padded \leftarrow total\_bytes/PAGESIZE$
$leftover \leftarrow total\_bytes\%PAGESIZE$
if $(leftover > 0)$
  $num\_pages\_padded \leftarrow num\_pages\_padded + 1$
$mem \leftarrow$ ALLOCATE_PAGES(num_pages_padded)
$address \leftarrow mem + pad * L1LINESIZE$
if $(pad + 1 >= L1LINESPERPAGE)$
  $pad \leftarrow 0$
else
  $pad \leftarrow pad + 1$

return $(address)$

2.8. Comments on results from other AMD Opteron systems

The performance loss when we get cache line contention is about 20% for the original code in Figure 1. On other Opteron systems we have seen as much as 50% performance loss, thus making it a very severe issue.

3. Intel Itanium-2

The 64-bit precision results for `Yee_bench` on the Intel 900 MHz Itanium-2 are shown in Figure 2. There is a clear correspondence between the performance and the L3 cache hit rate. The L2 cache size was 256 kbytes and the L3 cache size was 1.5 Mbytes. (The L1 cache is not used by floating-point data.)

The L3 cache is referenced whenever we have an L2 cache miss. A majority of the L2 cache misses are compulsory cache misses and therefore become L3 cache misses unless the problem size is small. As mentioned in Section 2.4, we have ten loads per iteration. Six of these loads are compulsory cache misses, two of these loads are highly likely to hit L2 cache, since they were used for the previous iteration value of the middle iteration. Two of the loads refers to values that were used in the previous iteration of the outer loop. These may hit L2 cache. If they miss L2, they might hit L3. For large problem sizes, they will miss both, and we will get an L3 cache hit rate that is almost zero. For $N = 100$ they will almost always miss L2 but hit L3 and we will get an L3 hit rate of 25% (two hits per six compulsory misses). The data traversed during one iteration of the outer loop is $64N^2$ bytes. For $N = 100$ this becomes 0.61 Mbytes, which is well within the L3 cache size.

The reason for the increase in performance when going from $N = 45$ to $N = 100$ is the loop restart penalty for the inner loop [2]. Examining the assembly code, we find that the *prolog* (and the *epilog*) consists of four iterations (see Chapter 2 in [3] for an explanation of the terms prolog and epilog). This is independent of N and is thus more expensive for smaller values of N.

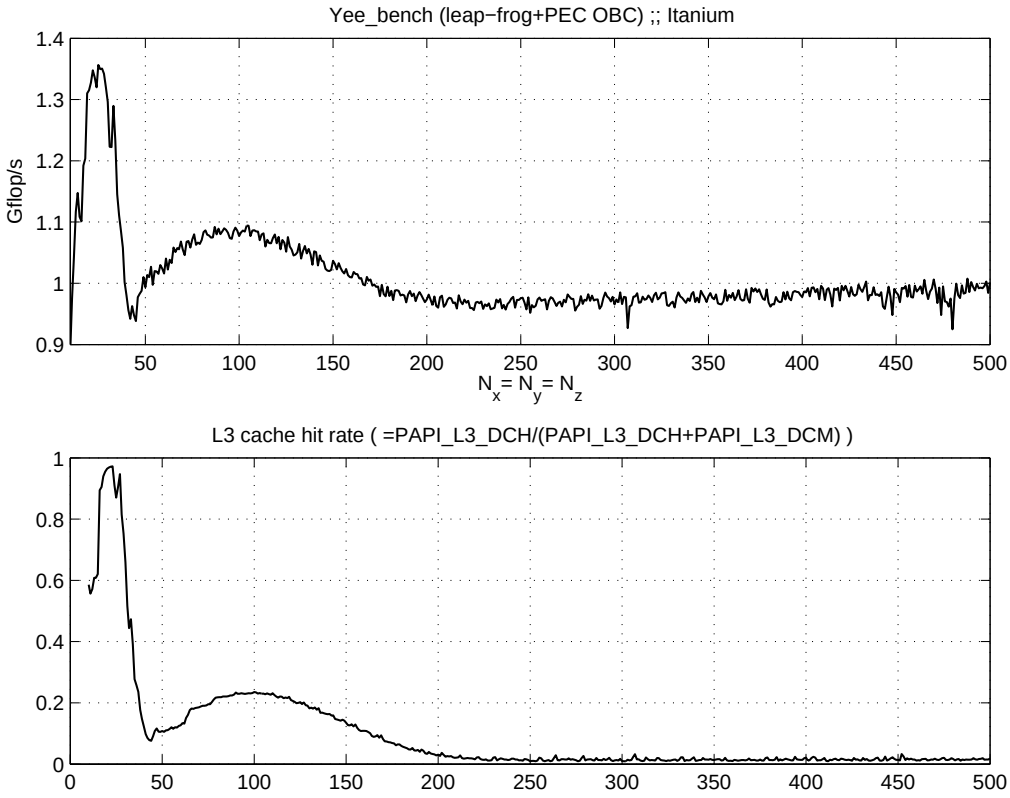


Figure 2. `Yee_bench` performance on the Intel Itanium-2.

4. Intel Xeon EM64

Figure 3 displays the 64-bit precision performance of `Yee_bench` on the Intel 3.4 MHz Xeon EM64. Results are compared for the original and the padded code. The padding used was `padEx=padHx=(/0,0,0/)`, `padEy=padHy=(/1,0,0/)`, and `padEz=padHz=(/2,0,0/)`. Again we see that padding smooths the performance curve, but results in a slight performance loss for medium sized ($30 < N < 100$) problems. Problem sizes smaller than $N = 28$ fits into the one Mbyte L2 cache for the unpadded code.

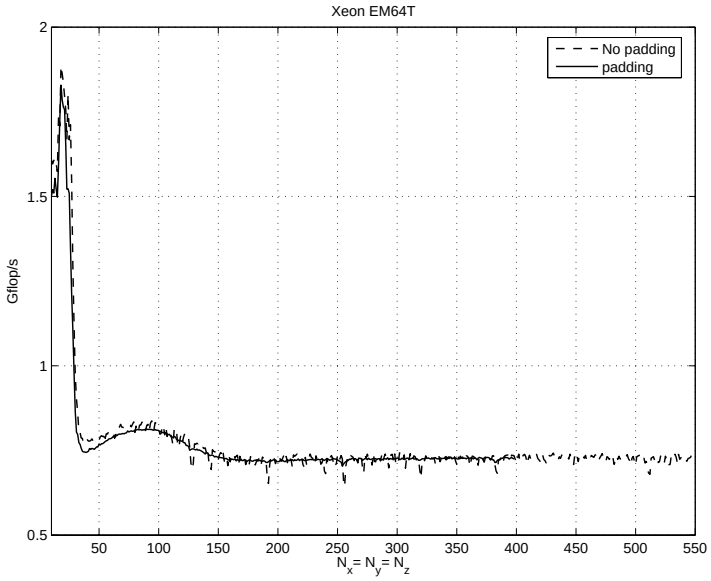


Figure 3. `Yee_bench` performance on the Intel Xeon EM64.

5. Variable Intra-Array Padding as a Technique for Padding Dynamic Data Structures

Stepping back, it is clear that what makes this problem interesting is that these arrays are allocated dynamically and thus we have no direct control over the starting address. As more and more HPC applications move to Fortran 90, traditional approaches like common block padding and power-of-two avoidance become obsolete. There is nothing special about `Yee_bench` that makes it amenable to the solution we have put in place here, except for compile time constants for variable amounts of padding. If a compiler were instead to do this work at allocate time, it would have to be able to discover array references with possible conflicts and then rewrite their corresponding allocate statements. This is only possible with significant work in an intra-procedural analysis phase, which some compilers lack entirely. Replacing `ALLOCATE`, as discussed in Section 2.7, is an option, but that requires knowledge of the compiler's run-time system and the exact arguments and linkage of the call to the `ALLOCATE` intrinsic. Instead of the above approaches, consider that the compiler could generate code for array declarations such that every array in the application was padded by some pseudo-random amount of at least a cache line. This would include both static and dynamic allocations. This could be taken a step further and pad the starting address of each array by a similar amount. This approach drastically reduces the chances of a way conflict. Whether done by compiler or by hand, we believe that variable padding scheme method is generalizable to other codes.

6. Comments on the Tuning Process

On of the important lessons to be learned here is that we were able to gather hardware performance data with `PapiEx` that directly correlated with the codes performance. As this was a benchmark, it contained internal timers to compute a theoretical GFLOPS number. The code could have just as easily reported its performance in terms of timesteps/day or cell-domains/second. What was important was that the code's performance metric and the metric chosen for analysis (L1 miss rate) were both rates and thus independent of problem size.

Another important point is that the fact that no source code instrumentation is needed to use `PapiEx` means that an investigation of strange performance can be started immediately and that the executable was not perturbed in any way through instrumentation.

A. The core of `Yee_bench`

The core of `Yee_bench` is two triple-nested loops, one for updating the magnetic field components and one for updating the electric field components. Close to 100% of the time is spent in these two triple-nested loops if the surrounding time-stepping loop contains enough time steps to dominate the initialization time. The code for the update of the magnetic field components is:

```
do k=1,nz ; do j=1,ny ; do i=1,nx
   Hx(i,j,k) = Hx(i,j,k) + (  (Ey(i,j,k+1)-Ey(i,j  ,k))*Cbdz +   &
                              (Ez(i,j,k  )-Ez(i,j+1,k))*Cbdy  )
   Hy(i,j,k) = Hy(i,j,k) + (  (Ez(i+1,j,k)-Ez(i,j,k  ))*Cbdx +   &
                              (Ex(i  ,j,k)-Ex(i,j,k+1))*Cbdz  )
   Hz(i,j,k) = Hz(i,j,k) + (  (Ex(i,j+1,k)-Ex(i  ,j,k))*Cbdy +   &
                              (Ey(i,j  ,k)-Ey(i+1,j,k))*Cbdx  )
end do ; end do ; end do
```

where `Cbdx` etc. are constants. The code for updating the electric field components is very similar.

References

[1] Ulf Andersson. Yee_bench—A PDC benchmark code. TRITA-PDC 2002:1, KTH, November 2002. Available at `http://www.pdc.kth.se/info/research/trita.html`.

[2] Ulf Andersson, Per Ekman, and Per Öster. Performance and performance counters on the Itanium 2 — A benchmarking case study. In G. R. Joubert et al., editors, *Parallel Computing: Software Technology, Algorithms, Architectures & Applications*, volume 13 of *Advances in Parallel Computing*, pages 517–524. Elsevier, 2004. Proceedings from ParCo2003.

[3] M. Cornea, J. Harrison, and P. T. P. Tang. *Scientific Computing on Itanium-based Systems*. Intel Press, 2002.

[4] John D. McCalpin. The `STREAM2` home page. `http://www.cs.virginia.edu/stream/stream2/`.

[5] Allen Taflove, editor. *Advances in Computational Electrodynamics: The Finite-Difference Time-Domain Method*. Artech House, Boston, MA, 1998.

[6] Allen Taflove. *Computational Electrodynamics: The Finite-Difference Time-Domain Method*. Artech House, Boston, MA, third edition, 2005.

[7] `PAPI` homepage. `http://icl.cs.utk.edu/papi/`.

[8] `PapiEx` homepage. `http://icl.cs.utk.edu/~mucci/papiex/`.

[9] Finite-Difference Time-Domain literature database. `http://www.fdtd.org/`.

Holistic Hardware Counter Performance Analysis of Parallel Programs

Brian J. N. Wylie[a], Bernd Mohr[a], Felix Wolf[a]

[a]John von Neumann Institute for Computing, Forschungszentrum Jülich, D-52425 Jülich, Germany

The KOJAK toolkit has been augmented with refined hardware performance counter support, including more convenient measurement specification, additional metric derivations and hierarchical structuring, and an extended algebra for integrating multiple experiments. Comprehensive automated analysis of a hybrid OpenMP/MPI parallel program is demonstrated with performance experiments, containing communication and synchronisation metrics combined with a rich set of counter metrics, which provide a holistic analysis context and facilitate multi-platform comparison.

1. Introduction

Modern microprocessors have integrated event counters which offer low-overhead access to a potential wealth of execution performance information, encompassing the utilisation and efficiency of various functional units and the memory and cache hierarchy. Although microprocessors from different manufacturers, and also within microprocessor families, provide broadly similar functionality, there are often very significant differences: variation in processor architecture and memory/cache hierarchy are reflected in corresponding event provision, and when combined with restrictions on which events may be measured simultaneously (and limited numbers of event counters) this greatly complicates performance measurement and analysis.

Various libraries have addressed the measurement issues, providing a portable application programming interface to event counter control and access (e.g., PAPI [6]). Along with interfacing to system libraries, these offer standardised definitions for the most important and universally available events, and mappings to the native events provided by each microprocessor. Additional events may be derived from one or more native events (if the processor supports their simultaneous measurement) and imposed counter time-sharing/multiplexing may provide a means for approximating the measurement of multiple counters within a single program execution. Although these approaches address the goal of acquiring a richer set of measurements in a particular experiment, it is notable that there is corresponding additional complexity which complicates interpretation. There may also be ambiguities in the definitions of events (such as whether speculative instructions are included in event counts or not) which must also be taken into account during their analysis.

Interpretation and analysis of performance counters has therefore been hindered, limited to a very small subset of the potentially usable events, and often specific to particular processor platforms. One goal of our current work has been to investigate the extent that it is possible to incorporate a wider range of counter metrics, both universal and platform-specific, and exploiting multiple measurement experiments where necessary, for holistic analysis of execution performance.

1.1. Initial KOJAK approach

Previous developments of the KOJAK performance measurement and analysis environment for parallel programs, which supports many current computer systems[1], offer a suitable vehicle for pursuing this investigation. KOJAK provides semi-automatic instrumentation of user applications and automatic analysis of performance problems arising from inefficient usage of parallel programming

[1]Download available from `http://www.fz-juelich.de/zam/kojak/`

interfaces (such as MPI and OpenMP) [1,2]. Performance problems are classified by type and quantified by severity, for investigation via an interactive browser (CUBE) which presents an integrated, hierarchical view of performance behaviour, call path and process/thread of execution.

A basic infrastructure also exists in KOJAK for measuring counter events and their incorporation into hierarchical analyses alongside communication and synchronisation metrics. One approach extended KOJAK's portable execution tracing to directly include counter measurements and incorporate them in its various analyses [3]. Another incorporates hardware counter analysis from separate platform-specific profiling tools with KOJAK's own execution trace analysis [4]. In both cases, counter measurements/metrics are related to program and system entities (i.e., the call tree, processes and threads) and quantified. While the second approach has a limited separated hierarchy of raw counter measurements, the first was an initial attempt to assess corresponding time penalties and integrate these with KOJAK's directly-measured time-based performance properties.

Quantifying time-penalties for event counts was promising, however, further investigation with additional metrics highlit the limitations of the approach. Where KOJAK identified a metric tuple (call-path and thread) with an occurrence rate above or below a certain threshold, it derived a performance penalty as the entire measured execution time of that tuple; in effect it used an upper bound on the actual penalty, for want of a better approximation. Comparing the derived performance penalties with those directly measured from cycles-based stall counters (on platforms which support them, e.g., UltraSPARC [9]), showed that while they were broadly representative, they were also significantly exaggerated. In this case, the measured penalties could have been used to adjust the performance penalty derivations to improve their accuracy, though the derivations would inevitably be platform-specific (and it would generally not be possible to quantify the actual penalties). Furthermore, the performance of a tuple is ultimately due to multiple causes, manifesting in multiple counter metrics and also non-counter metrics (e.g., communication and synchronisation times), in complex dynamic relationships, such that it is not possible to accurately determine the time penalty related to a single count measurement. Although the exaggeration of particular performance aspects can be broadly in-line with their actual severity, and as such benefit analysis, in practice it was found to have a detrimental impact on the analysis as a whole, by subtly compromising its integrity.

2. Refined design for hardware counter measurement and analysis

A more robust foundation for incorporating event counts from hardware counters into performance experiments is to integrate them in separate metric hierarchies presented alongside that for measured time metrics. This is particularly the case when larger numbers of counters are measured for analysis. Since it is rare that processors support simultaneous measurement of all of the counters of interest, multiple measurements with subsets of counters may be required, with these partial experiments integrated into a single comprehensive analysis. Assistance can also be provided with specification of appropriate sets of counters for measurement, and multiple presentation hierarchies may be valuable during analysis.

These various aspects have been addressed to refine KOJAK support for counter-based analysis within the existing framework of MPI and OpenMP communication and synchronisation analysis.

2.1. Structured analysis via metric hierarchies

Defining hierarchies of related counter events both provides an improved structure for navigating and interpreting the relationships between events (such as data references encompassing loads and stores, or hits and misses at different levels of cache and memory) and assessing their significance (e.g., cache misses as a proportion of references). In some cases, it can be clear that a single natural

hierarchy of related events can be defined. Generally, however, a set of event data may profitably be structured in several hierarchies, where it may not be possible to determine in advance which is most valuable: indeed, the various hierarchies are often complementary rather than redundant. Furthermore, while part of a hierarchy may be platform/processor-independent, it is desirable to be able to include available platform/processor-specific events for a more complete and detailed understanding of execution performance, which itself may well be platform-specific.

For example, consider the hierarchy of caches used to improve the performance of data accesses from memory. A general categorisation of data (and instruction) accesses uniquely associates them with the level of cache or system memory from which they are provided, i.e., where they hit:

```
DATA_ACCESS = DATA_HIT_L1$ + DATA_HIT_L2$ + ... + DATA_HIT_MEM
```

It can also be inferred that misses occurred in lower levels of cache. Data accesses to each level can be reads/loads or writes/stores, offering the next general division:

```
DATA_HIT_L1$ = DATA_LOAD_FROM_L1$ + DATA_STORE_INTO_L1$
```

It is worth noting that this general hierarchy, while applying to a variety of processors and systems, contains elements which will not apply on all: e.g., IBM p690+/POWER4-II [7] has three levels of cache whereas Opteron [8] and UltraSPARC-III/IV [9] only have two, and while the latter can register stores into each level of cache (and memory) the former only registers stores into L1 cache which write-through to the rest. This is readily handled with the proposed structuring, as the inapplicable L3 cache measurements can be treated as zero-valued (i.e., equivalent to a non-functional L3 cache).

Provision of hardware counters also varies considerably by processor/system. Opteron has a counter to measure data accesses directly, so an Opteron-specific definition can be used,

```
DATA_ACCESS = DC_ACCESS                                                  # Opteron
```

however, data accesses must be derived from the *composition* of other events on UltraSPARC-III/IV and POWER4-II, and such composed metrics are fundamental to the hierarchical structure. L1 cache read and write hits can not be measured directly by the UltraSPARC or POWER4-II counters, however, they can be determined by a *computation*[2] with measured counters:

```
DATA_LOAD_FROM_L1$ = DC_rd - DC_rd_miss                                  # US-3/4
DATA_STORE_INTO_L1$ = DC_wr - DC_wr_miss                                 # US-3/4
DATA_LOAD_FROM_L1$ = PM_LD_REF_L1 - PM_LD_MISS_L1                        # POWER4
DATA_STORE_INTO_L1$ = PM_ST_REF_L1 - PM_ST_MISS_L1                       # POWER4
```

Opteron doesn't provide counters which can distinguish L1 cache read and write hits, or even allow their combination to be measured directly, however, this can also be computed instead:

```
DATA_HIT_L1$ = DC_ACCESS - DC_MISS                                       # Opteron
```

While such computed metrics provide a valuable means for completing the general hierarchies, when compositions are not available, they don't provide the benefit of extending the hierarchies in the way that composed metrics naturally do. For example, data load hits from L2 cache are composed from multiple native events on Opteron and POWER4-II, respectively:

```
DATA_LOAD_FROM_L2$ = DC_L2_REFILL_O + DC_L2_REFILL_E + DC_L2_REFILL_S # Opt
DATA_LOAD_FROM_L2$ = PM_DATA_FROM_L2
                   + PM_DATA_FROM_L25_MOD + PM_DATA_FROM_L25_SHR
                   + PM_DATA_FROM_L275_MOD + PM_DATA_FROM_L275_SHR # POWER4
```

[2]The term *computation* is defined as a general calculation which can include subtractions (and potentially other arithmetic operations), whereas *composition* is defined to be strictly additive.

Although these compositions have quite different constituent measured counters, they naturally extend the general hierarchy with additional platform-specific detail, which can offer further insight for performance tuning on the respective platforms. While each (dual-core) POWER4-II processor has its own local L2 cache, it shares this with the other processors on its multi-chip module (MCM, L25) and the processors on the other MCMs in its node (L275), all of which are faster than accessing L3 cache (which is similarly shared), so local versus remote L2 cache accesses impact performance.

This process of deriving hierarchies of new metrics from compositions and computations of available measurements is able to create quite comprehensive structured relationships for data, instruction and TLB accesses (and associated hits and misses), with a general structure extended by additional platform-specific components. Metrics which are not applicable, or can't be derived from available measurements can be omitted. When a composition is only partially satisfied by available measurements, it can still be valuable to retain it, but it should be clearly indicated as incomplete, such as including '$\sim$' in its label. (Where a particular set of measurements include such partially satisfied derivations, these may subsequently be completed when experiments are combined.) Partial computations can have negative values or values in excess of their parent, such that it's generally not prudent to retain them: in most cases, measurements can be grouped such that those required for computed metrics are kept in the same group to avoid this.

Similar structuring can also be applied to the types of instruction processed by various functional units and cycles-based counters for related busy/stall and idle periods. In these cases, more of the measurements are platform-specific and while it's still possible to have a hierarchical relationship, there are typically more 'gaps' corresponding to unmeasurable/unaccounted events. There can also be considerable ambiguity regarding particular events and the counters which measure them. For example, since storing floating-point data is typically done by the floating-point unit (FPU), this is often naturally accounted as a floating-point event: where this is not desired, the corresponding event measurement can be relocated to another category, such as `MEMORY`. Often, however, it may not be possible to distinguish the different kinds of events counted by particular functional units. There may also be inconsistency between counting instructions issued and those which actually complete.

While a general classification and hierarchy of a variety of processor events can be developed, it is ultimately necessary to refer to the respective processor manuals (and associated documentation of native counter events) to assess their significance [7–9].

2.2. Flexible metric specification and customisation

Metric structuring which specifies (presumed) relationships between events provides a mechanism for helping to navigate and understand those relationships. While generic hierarchies such as those described offer one particular structuring, alternative or complementary structures may also be defined and preferable in some cases. Measured events which fit no hierarchy must simply be listed separately (as is the case when no relationships are associated with a metric).

A flexible approach is therefore taken, which provides the specification of metric relationships in a text file which is read to configure and structure the analysis: specifications shown in the previous subsection are extracts from such a file. The default specification can then be overridden to provide alternative analyses when desired. A specification file also offers convenience during measurement collection, providing definitions of groups of counters which can usefully be collected in the same measurement, i.e., taking into account restrictions on the number and types of events that can be counted simultaneously. Although it is possible to use PAPI preset names for counters to create notionally-portable groups, it is preferable to specify platform-specific groups directly in terms of native events, since many of the relevant native events have no corresponding PAPI preset definition and combination of presets is still subject to the same platform-specific limitations.

2.3. Holistic analysis via integration of multiple experiments

Analysis of hardware counter measurements, and metric derivations therefrom, can take two broad approaches. The first sticks strictly to what can be reliably determined from a single measurement experiment (as is the case for HPM [7] and Apprentice^2 [10]), and as such is significantly limited by the flexibility and capabilities of the actual monitoring hardware provided by the processor. Several, separate experiments with different sets of measurements may be considered, with the implicit understanding that the execution may be quite different in each case. An alternative uses time-sharing or multiplexing to automatically change the events measured throughout the duration of an experiment, and extrapolate from these partial measurements to a larger set of approximate measurements. Whereas this has the convenience and benefit of handling a single execution, it can be compromised by variations in behaviour within the execution (though these may be small if the execution is sufficiently regular and long with respect to the time-sharing period).

Requiring multiple executions is a significant overhead, however, it also provides an opportunity to consider possible run-to-run variations and incorporate them in the analysis. While past results are no guarantee of future performance, they can help indicate what range of performance can reasonably be expected. This is particularly useful for deterministic applications when the hardware configuration is unchanged and executions occur in a relatively controlled (dedicated) environment.

KOJAK's CUBE algebra operators [1] allow experiments to be combined to produce the mean of multiple related experiments or to aggregate experiments containing different hardware counter metrics. Combining both approaches can be used to reduce run-to-run variations and extend the metric analyses to the set of experiments. Furthermore, the difference of two experiments can be calculated to examine variations between them.

The existing merge utility produced an experiment with the union of metrics, call-paths and process/thread measurements in input experiments. This was extended to integrate experiments containing identical call-path and process/thread trees, but different sets of measured and derived hardware counter metrics. Measurements replicated in more than one experiment are averaged, however, measurements contributing to metric compositions, and which are only partially fulfilled in individual experiments, are accumulated to allow the compositions to be completed. Where available, measured metric values are also retained in preference to partially computed or accumulated values.

3. Results

To demonstrate these new KOJAK capabilities, three comprehensive sets of experiments consisting of complementary groups of hardware counter measurements were collected on an IBM Regatta cluster, Cray XD1 cluster and Sun Fire E25000, using the ASC Purple sPPM v1.1 benchmark [11]. This application uses a simplified piecewise parabolic method (PPM) to solve a 3D gas dynamic problem on a uniform Cartesian mesh. It is written mostly in Fortran 77 and can simultaneously exploit multithreading for shared-memory parallelism and domain decomposition with message passing for distributed parallelism: the double-precision (64-bit) hybrid parallelisation tested used 32 MPI processes each with 2 OpenMP threads. The processes were partitioned $2 \times 4 \times 4$ in the $X \times Y \times Z$ dimensions, a configuration chosen to offer a reasonably close comparison between the experiments on the different systems, rather than being optimised for any particular system.

Preparation of the instrumented application executables was done by prepending kinst-pomp to the commands that invoke the compiler and linker. This runs a source preprocessor to automatically instrument the application's 12 OpenMP parallel DO loops, 41 explicit barriers and various additional single and master blocks, and link instrumented PMPI and POMP libraries along with the PAPI library for hardware counter measurements. To provide additional context for the analysis,

while avoiding overheads associated with automatically instrumenting the entry and exits of every application routine, the program's main phases and the key routines using MPI and OpenMP had also previously been manually annotated with POMP region instrumentation directives [5]. When the instrumented applications are executed in the usual fashion (and with optional hardware counter measurements configured through an environment variable), the instrumented events are recorded in per-thread trace buffers which are subsequently merged into single traces for each execution.

The experiments used two p690+ nodes of an IBM Regatta cluster (running AIX 5.2 and connected via HPS) consisting of 4 MCMs with 4 dual-core POWER4-II processors, 32 nodes of a Cray XD1 cluster (running GNU/Linux 2.6 and connected via RapidArray network) each with two AMD Opteron 248 processors, and a Sun Fire E25000 (running Solaris 9) with dual-core UltraSPARC-IV processors. On the IBM system, 6 experiments were collected (with up to 8 counters in each), whereas 10 experiments (each with 4 counters) on the XD1 and 18 experiments (each with 2 counters) on the E25000 were required to acquire a comparable level of detail. These sets of experiments were subsequently incorporated into a single composite analysis experiment for each platform.

3.1. Comparative experiment analysis

For this analysis, execution times (and other absolute measurements) are less important than relationships between measurements, whether within a set of experiments or between sets: Figure 1 shows a view of the analysis of the XD1 and Regatta experiments. Due to space limitations, the E25000 experiment is not included in this abbreviated analysis: for additional analyses see [12].

Wall-clock execution time of 180s (163s in the runhyd computational kernel) on the XD1 compares with 280s (241s in runhyd) on the Regatta for each experiment. Parallel initialisation overheads (in the init phase) amount to 1.9% of execution time on the Regatta versus 0.5% on the XD1, with the balance attributed predominantly to the runhyd computational kernel, within which the six routines responsible for the hydrodynamics each account for roughly equal shares of the total, and each has good load balance over the 64 threads (32 processes) on both platforms.

The respective proportions of total execution time attributed to MPI are very similar — 1.7% on XD1 vs. 2.2% on Regatta — and investigating further, this corresponds primarily to point-to-point communication, with (the master threads of) every fourth process responsible for contributing twice as much as the others. The MPI_Allreduce in glblmax at the end of the main computation loop in runhyd can also be found to require a significantly higher collective wait time on the Regatta, totalling 127s (0.77%) versus 15s (0.15%) on the XD1.

OpenMP runtime costs on the XD1 are attributed 3.3% of total execution time, versus 0.9% on Regatta, further categorised as explicit barrier synchronisation wait time in each case. Whereas this is mostly attributed to the six hydrodynamics routines on the XD1, with only 4% in the barrier at the end of the computational loop, on the Regatta that final barrier is attributed 82%.

Some potentially important differences in the MPI and OpenMP communication and synchronisation can therefore be seen in the XD1 and Regatta experiments, however, they also demonstrate broadly similar parallelisation efficiency. Proceeding beyond the parallel execution, communication and synchronisation times, additional performance metrics are provided by and derived from hardware counters measurements. While subsets of the counter-based metrics are available in individual experiments, in combination they offer comprehensive insight into the processors' execution.

Comparing the proportion of mispredicted branches (*BRANCH_MISP*), while relatively small in both cases, at 0.58% of all instructions (7.0% of branches) it is considerably larger for Regatta than the 0.08% (1.5%) of Opteron, and depending on the selected call-path is also seen to vary considerably by thread, with some threads notably more affected than the others. The significance can be investigated further by examining the respective counters which measure branch stall cycles.

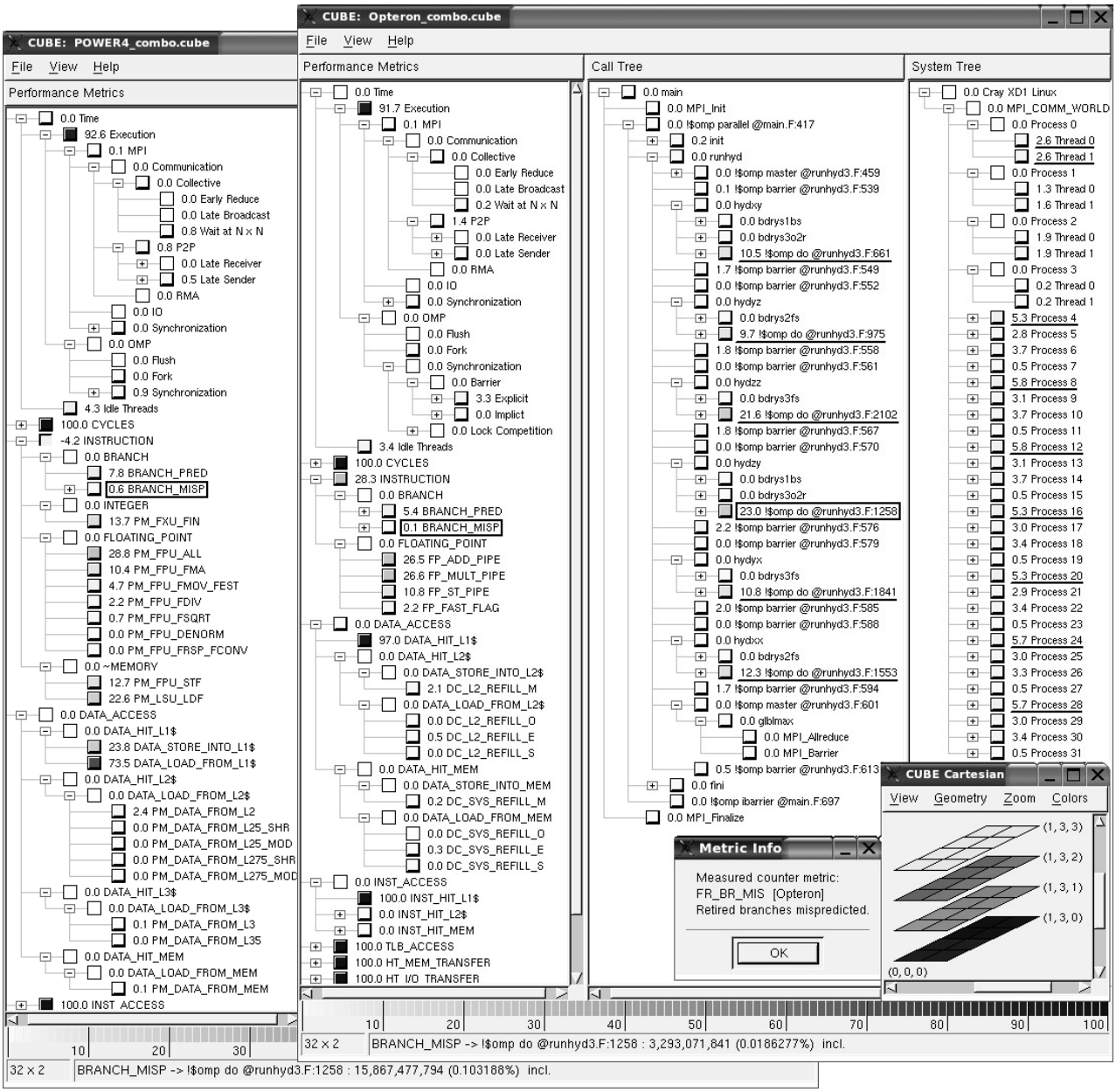


Figure 1. Two KOJAK analyses of combined hybrid OpenMP/MPI sPPM benchmark executions on equisized Cray XD1 Opteron and IBM Regatta p690+ POWER4-II clusters (in front and below). Performance metrics (left pane) and their distribution over the program's call tree (middle pane) and process/thread tree (right pane) are presented hierarchically. Metric values have been expressed as percentage of total execution time or root counter value, and shown with squares coloured according to the scale at the bottom. Selectively expanding or collapsing nodes in each of the three linked trees allows analysis at different levels of granularity. The currently selected metric for branch misprediction rates (with its derivation visible in the pop-up) and the call-tree path ending with the parallel loop in one of the six key hydrodynamics routines are shown boxed (and corresponding details provided in the area at the bottom). Important processes/threads and other call-paths are shown underlined and appear darkest in the lower right virtual process topology display. Each set of experiments has identical call-tree and system tree hierarchies, and only performance metrics which are platform-specific counters differ, yet the broad similarity facilitates comparisons between them.

Both processors are seen to have 97% of data accesses hit L1 cache, however, it is the increasingly costly accesses that miss L1 cache and must be satisfied from higher caches and memory that are most significant and warrant further investigation. On p690+ these are seen to be predominantly from local L2 cache (`PM_DATA_FROM_L2`), with only 0.14% requiring to come from memory. With its smaller, two-level caches, Opteron must load twice as much (0.27%) of its data from memory.

4. Conclusion

Refinement of KOJAK's hardware-counter-based analysis retained much of the existing measurement, recording and analysis infrastructure, with the incorporation of functionality for more convenient counter-metric measurement specification, additional metrics derivable from measured metrics, and customisable structured metric hierarchies. Furthermore, the algebra for integrating multiple experiments was extended to consolidate experiments containing (sub)sets of counter-based metrics and produce unified experiments with all of the available measured and derivable metrics.

Unified experiments, containing communication and synchronisation metrics combined with a rich set of counter metrics, support comprehensive holistic analysis of parallel programs: execution inefficiencies may be isolated to particular processors (or threads) and their various functional units, or found to relate to the use of shared and distributed caches and memory within modern computer systems. The portable CUBE format of analyses also allow fuller comparison between platforms, where architectural differences may be significant. These capabilities contrast those of existing tools which can also offer detailed platform-specific analysis when appropriately directed by knowledgable users, but without a holistic overview and context, or multi-platform comparison.

References

[1] Felix Wolf and Bernd Mohr: "Automatic Performance Analysis of Hybrid MPI/OpenMP Applications," J. Systems Architecture, 49(10–11):421–439, Elsevier, Nov. 2003.

[2] Felix Wolf: "Automatic Performance Analysis on Parallel Computers with SMP Nodes," PhD dissertation (RWTH Aachen, Germany), NIC Series, Vol. 17, Forschungszentrum Jülich, 2003.

[3] Felix Wolf and Bernd Mohr: "Hardware-Counter based Automatic Performance Analysis of Parallel Programs," Proc. Conf. on Parallel Computing (ParCo'03, Dresden, Germany), Parallel Computing: Software Technology, Algorithms, Architectures & Applications, pp. 753–760, Elsevier, 2004.

[4] F. Song, F. Wolf, N. Bhatia, J. Dongarra, and S. Moore: "An Algebra for Cross-Experiment Performance Analysis," Proc. Int'l Conf. on Parallel Processing (ICPP'04, Montreal, Canada), pp. 63–72, Aug. 2004.

[5] B. Mohr, A. Malony, H.-C. Hoppe, F. Schlimbach, G. Haab, J. Hoeflinger, and S. Shah: "A Performance Monitoring Interface for OpenMP," Proc. 4th European Workshop on OpenMP (Roma, Italy), Sept. 2002.

[6] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci: "A Portable Programming Interface for Performance Evaluation on Modern Processors," Int'l J. HPC Applications, 14(3):189–204, 2000.

[7] Luis A. DeRose: "The Hardware Performance Monitor Toolkit," Proc. 7th Int'l Euro-Par Conf. (Manchester, UK), Lecture Notes in Computer Science, Vol. 2150, pp. 122–131, Springer-Verlag, Aug. 2001.

[8] Advanced Micro Devices, Inc.: "BIOS and Kernel Developer's Guide for AMD Athlon 64 and AMD Opteron Processors," Pub.#26094, Rev. 3.14, Apr. 2004.

[9] Sun Microsystems, Inc.: "UltraSPARC Processors," `//www.sun.com/processors/manuals/`

[10] Cray, Inc.: "Cray Performance Analysis Tool-set (PAT & Apprentice^2)," `/opt/xd-tools`, Feb. 2005.

[11] John Engle: "The ASC Purple sPPM Benchmark Code," Lawrence Livermore National Laboratory, USA `//www.llnl.gov/asc/purple/benchmarks/limited/sppm/`, Feb. 2002.

[12] Brian J. N. Wylie, Bernd Mohr, and Felix Wolf: "Holistic Hardware Counter Performance Analysis of Parallel Programs," Technical Report FZJ-ZAM-IB-2005-14, Forschungszentrum Jülich, Oct. 2005.

Ring algorithms on heterogeneous Windows-based clusters with various message passing environments

Andrea Clematis [a], Angelo Corana [b]

[a]IMATI-CNR, Via De Marini 6, 16149 Genova, Italy

[b]IEIIT-CNR, Via De Marini 6, 16149 Genova, Italy

The aim of the present work is the development of efficient and self-adaptive ring algorithms on heterogenous Windows-based clusters. We show that a virtual ring of processes, with a number of processes on each node proportional to its relative speed, greatly reduces load imbalance and allows to achieve good performance even on highly heterogeneous systems.

As test application we consider the computation of long- and short-range interactions.

Two different implementations of MPI for Windows are considered (MPICH and MPICH2) and some comparisons with PVM are also performed. The analysis is quite general and can be applied to similar problems. From the algorithm analysis we obtain both a full computer simulator of ring applications and some simplified indices of performance, useful to quickly adapt the application to a given platform.

1. Introduction

In recent years clusters of PCs for parallel computing have become very popular owing to several favourable characteristics, namely good cost to performance ratio, availability, flexibility.

The increasing power of nodes and speed of interconnecting network [1] allow to build very powerful machines, able to compete with traditional high performance computers. Recently, PC clusters and/or networks of PCs are evolving towards the emerging Grid architecture (desktop and global grids) or they can be viewed as components of a larger grid. Although most of network-based parallel computing systems run some Unix O.S. (in particular Linux), the systems based on Windows O.S. are also interesting, since they allow to exploit for parallel applications the great number of Windows machines available in various organizations, both scientific, business and industrial.

In this work we consider the development and the analysis of efficient and self-adaptive ring algorithms on heterogeneous clusters (i.e. with nodes of different speed). It is well know that standard ring algorithms, with one process per node, are perfectly balanced on homogeneous and dedicated parallel systems [2], but they give poor performance on heterogeneous, possibly non dedicated, resources. In particular, we show as a virtual ring of processes, with data evenly partitioned among processes, allows to obtain very good performance also on highly heterogeneous platforms.

As test application we consider the computation of long- and short-range interactions. In particular, we deal with the computation and histogram of distances (all distances or just those involving neighbouring pairs) in a large set of points in R^m. Such test problem is interesting, since varying the neighbour size we are able to vary the computation to communication ratio.

After the computational analysis of this approach, which gives us both a full computer simulator and some simplified indices of performance, we present and compare some experimental results collected on a heterogeneous platform, namely a cluster of PCs with Windows O.S.

To implement the application, we consider two freely available implementations of MPI for Windows: MPICH NT 1.2.5 and MPICH2 1.0-1. As comparison, we also consider the version for Windows of the older but still popular PVM library (PVM v. 3.4.5).

2. The heterogeneous computing system

Let us consider a heterogeneous parallel system consisting of p nodes, connected by a switched communication network (e.g. Fast Ethernet, Gigabit Ethernet). Our analysis also applies to mixed networks.

Let be s_i the relative speed of node i with respect to a reference machine [3], e.g. the lowest machine in the set. s_i depends both on the processor features and on the application under consideration, and can only be estimated in an approximate way. In particular, it can depend on the amount of local data.

We suppose that the system is dedicated or that the load of each node remains nearly constant during the whole computation. So, time variation of speeds during computation is negligible.

The total relative speed of the system is $S = \sum_i s_i$ and it is also the ideal speed-up.

The speed-up and the global efficiency are [3]:

$$SU = \frac{T^{seq}}{T^{par}} = \sum_i s_i \eta_i, \qquad \eta = \frac{SU}{S} = \frac{\sum_i s_i \eta_i}{S} \tag{1}$$

where: T^{seq} and T^{par} are the execution time on the reference node and the parallel execution time on the heterogeneous system, respectively; $\eta_i, \quad i = 1, \ldots, p$ are the node efficiencies.

The degree of heterogeneity can be expressed in various ways; for ring applications we find useful the index $h = 1 - s_0/\overline{s}$, where $\overline{s}$ is the average relative speed and s_0 denotes the lowest relative speed in the system $(0 \leq h < 1)$.

3. The computational problem

To test the proposed method we employ an application that we originally developed for homogeneous systems [4], and that we subsequently used with various computing platforms. It can be considered a kind of high level benchmark.

Given a set X of N points in R^m, the algorithm carries out the computation and histogram of distances between neighbouring pairs [2,4], i.e. pairs whose distance is less than a predetermined threshold ϵ, expressed as a fraction of the diameter of the pointset. As a particular case, if $\epsilon = 1$, all distances are computed. For the fast search of neighbouring pairs we use the box-assisted approach described in [4], in which an auxiliary m'-dimensional mesh of boxes, with $1/\epsilon$ boxes along each dimension, is used to build linked lists of points falling into the same m'-dim box.

The sequential execution time on the reference node is

$$T^{seq} = \left(\frac{N \cdot (N-1)}{2} \cdot f(\epsilon) \right) \cdot \tau_o(m) \tag{2}$$

where $\tau_o(m)$ is the CPU time to process a pair on the reference node, and $f(\epsilon) \leq 1$ is the fraction of the total pairs processed.

Our application is split into a virtual ring of $q \geq p$ processes, with q_i (logically neighbouring) processes on the i-th node (Fig. 1).

The set of points is evenly partitioned into q subsets X_j, of size $N' = \frac{N}{q}$ and each subset is assigned to a process. The total number of distinct pairs is decomposed in the following way

$$\frac{N \cdot (N-1)}{2} = q \left(\frac{N' \cdot (N'-1)}{2} + \frac{q-1}{2} \cdot N'^2 \right). \tag{3}$$

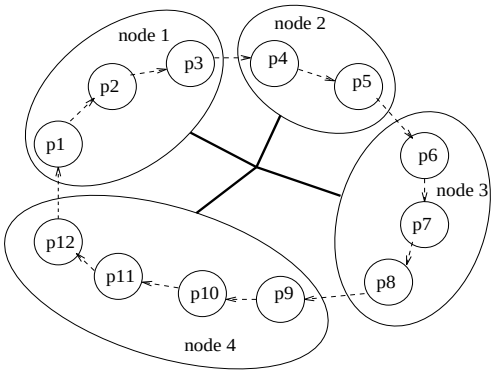


Figure 1. A virtual ring of 12 processes mapped on 4 nodes connected by a switched network.

The computation comprises two phases (Fig. 2): in phase one, which does not need communications, local pairs are processed; in the second phase, which consists of $L = \frac{q-1}{2}$ steps, local points of the generic process are moved forward along the ring in order to process pairs formed by local and visiting points.

Proc$_j$:
compute distances $\leq \epsilon$ in (X_j)
for $l = 1, L$
send data to next process
receive data (X_r) from previous process
compute distances $\leq \epsilon$ in (X_r, X_j)
endfor

Figure 2. Process structure ($1 \leq j \leq q$).

For each process, the CPU times (on the reference node) spent in phase 1 and in each step of phase 2 are:

$$t_1^{cp} = \frac{N'(N'-1)}{2} \cdot f(\epsilon) \cdot \tau_o, \qquad t_2^{cp} = N'^2 \cdot f(\epsilon) \cdot \tau_o. \tag{4}$$

If q is odd the computation is terminated; if q is even a further phase is needed in which $q/2$ processes receive points from their opposite. In the end all the partial histograms are summed. The fraction of total computation carried out in phase 1 $(1/q)$ becomes negligible as q increases.

As ϵ decreases $f(\epsilon)$ decreases, and the computation to communication ratio worsens.

We assume that the various subsets are statistically equivalent, i.e. $f(\epsilon)$ is the same for all subsets, resulting in a constant time to process any pair of subsets.

4. Performance analysis and modeling

The elapsed time on each node is the sum of computation time, context switching overhead, time needed for local activities involved with communications, and idle time (in general due to transmission time and imbalance).

We assume that, for each step in phase 2, the transmission time over the network between nodes i and j can be modeled as:

$$t_{i,j}^{tm} = \alpha_i + \beta_{i,j} \cdot size(N'm) \tag{5}$$

where α_i is the latency, $\beta_{i,j}$ is the communication time per byte, and $size(N'm)$ is the number of bytes to be moved. Similarly, using suitable parameters $\alpha_0, \beta_0, \alpha_{pk}$ and β_{pk}, we can model the intra-processor communication time t_0^{tm}, and the time for packing/unpacking $t^{pk} = t^{upk}$.
t_0^{tm}, t^{pk} and t^{upk} refer to the reference node, and are assumed to vary among nodes as $1/s_i$.

If N' is large it can be convenient to split data into strips of suitable size.

The cumulative elapsed time $T_{i,l}$ on the i-th node at the end of the l-th step is:

$$\begin{aligned} T_{i,0} &= t_i^1 \\ T_{i,l} &= T_{i,l-1} + t_{i,l}, \quad l = 1, \ldots, L \end{aligned} \tag{6}$$

t_i^1 is the elapsed time on the i-th node at the end of the local phase and $t_{i,l}$ is the elapsed time on the i-th node for the l-th step:

$$\begin{aligned} t_i^1 &= q_i \frac{t_1^{cp}}{s_i} + t_i^{cs1} \\ t_{i,l} &= \frac{q_i}{s_i}(t_2^{cp} + t^{pk} + t^{upk}) + t_i^{cs2} + \frac{(q_i - 1)}{s_i} t_0^{tm} + t_{i,l}^{idle} \end{aligned} \tag{7}$$

t_i^{cs1} and t_i^{cs2} denote the time lost due to context switching on the i-th node in the local phase and in one step respectively.

$t_{i,l}^{idle}$ is related to imbalance, and is computed at each step in the following way:

$$\begin{aligned} t_{i,l}^{idle} = \max(\; & T_{i-1,l-1} + \frac{q_{i-1}}{s_{i-1}}(t_2^{cp} + t^{pk}) + \frac{(q_{i-1} - 1)}{s_{i-1}}(t_0^{tm} + t^{upk}) + t_{i-1}^{cs2} + t_{i,j}^{tm} \\ & - T_{i,l-1} - \frac{q_i}{s_i}(t_2^{cp} + t^{pk}) - \frac{(q_i - 1)}{s_i}(t_0^{tm} + t^{upk}) - t_i^{cs2}, \;\; 0 \;). \end{aligned} \tag{8}$$

The parallel execution time is therefore $T^{par} = \max_i T_{i,L}$.

This formulation allows us to implement a computer simulator, whose accuracy depends on the number of overhead sources we consider.

Considering the CPU times at the three levels (step l on node i, node i, whole system) and the corresponding elapsed times, we can obtain [3] the efficiencies $\eta_{i,l}, \eta_i$ and η. The system wide efficiency is related to the node-level quantities by eq. (1).

The imbalance at the step level for the i-th node can be defined as

$$U_{s,i} = t_2^{cp}(q_{i_o}/s_{i_o} - q_i/s_i) \tag{9}$$

where i_o denotes the node with the highest CPU time ($q_{i_o}/s_{i_o} = \max_i(q_i/s_i)$).

Since imbalance is the main factor that limits performance for ring applications on heterogeneous systems, it is interesting to obtain in a approximate way the efficiency loss at the node level due to imbalance as

$$\gamma_i = \frac{U_{s,i}}{t_2^{cp}(q_i/s_i) + U_{s,i}} = 1 - \frac{s_{i_o}}{q_{i_o}} \frac{q_i}{s_i}. \tag{10}$$

Using eq. (1) we obtain the system wide quantity

$$\gamma = 1 - \frac{s_{i_o}}{q_{i_o}} \frac{q}{S}. \tag{11}$$

Following this approach, given an heterogeneous system and given the total number of processes q, the optimal allocation of processes is the one that minimizes $U_{s,i}$. This allocation can be found using dynamic programming [5], or, in a simpler but sub-optimal way, setting $q_i \simeq \frac{s_i}{S} q, \quad i = 1, \ldots, p$, and approximating fractional results to the nearest integer.

5. Experimental and simulated results

5.1. The PC cluster and the message passing environments

The application is tested on 5 different configurations selected in a cluster composed of 7 PCs (see Table 1) with Windows 2000 O.S., connected by switched Fast-Ethernet. The relative speeds s_i are average values since they present some fluctuations both with N' and with ϵ.

Table 1
Description of the various nodes in the cluster; a is the reference node

Node Id.	Type	Memory Size (MB)	s_i
a	PIII 600 MHz	128	1.00
b	PIII 800 MHz	128	1.20
c	PIII 866 MHz	256	1.30
d	PIII 1.3 GHz	256	1.80
e	PIV 1.8 GHz	256	2.07
f	PIV 2.4 GHz	256	3.28
g	PIV 2.8 GHz	512	3.59

To implement the application, we consider two message passing environments: MPICH NT 1.2.5 and MPICH2 1.0-1. They are the versions for Windows of the freely available and portable implementations of MPI (MPICH [6]) and MPI-2 (MPICH2 [7]). As comparison, we also consider the version for Windows of the popular PVM library (PVM v. 3.4.5) [8,9], older but still largely used for its simplicity and effectiveness. We use the Compaq Visual Fortran compiler v.6.6. We performed some preliminary point-to-point communication trials with the three environments, whose results are summarized in Table 2.

Table 2
Comparison of point-to-point communication speeds for the three message passing environments

Environment	$\alpha(\mu s)$	$\beta(ns/byte)$	$\alpha_o(\mu s)$	$\beta_o(ns/byte)$
MPICH2	30-120	90	10-45	11-43
MPICH	38-130	90	4-12	1-6
PVM	5000	300	5000	40-300

It results that communications in PVM are quite poor, since communication speed is limited by the various software overheads at the O.S. and PVM levels, resulting in a high latency and in a sustained rate well below the physical limit; moreover, intra-processor communications are not much faster than inter-processor communications [1].

Communications in MPICH and MPICH2 are significantly more efficient; moreover MPICH2 yields slightly faster inter-processor communications but slower intra-processor communications. Indeed MPICH uses sockets for inter-processor communications and shared memory for intra-processor communications, whereas our implementation of MPICH2 always uses sockets. In order to assure the maximum speed we implement ring communications in MPICH and MPICH2 using the buffered send (BSEND).

5.2. The procedure

We consider different problem sizes and various ϵ values. Points in R^m are obtained from the Henon time series [4] and normalized to the unitary hypercube; we choose $m = 10$, $m' = 2$ and use the euclidean norm. The time per pair measured on the reference node (PIII 600 MHz) is $\tau_o = 0.57\mu s$ and the $f(\epsilon)$ values for our data set are: $f(1) = 1$, $f(2^{-3}) = 0.23$, $f(2^{-6}) = 0.023$, $f(2^{-9}) = 0.0051$.

The trials are executed on dedicated nodes and with a low traffic on the network.

Since our aim is the development of self adaptive applications [10], able to maximize performance for a given hardware configuration, we employ the following procedure.

1) At the beginning we execute on the target system a small instance of the computation, with one process per node, just to evaluate the actual node speeds.

2) Depending on the measured s_i, a suitable number q of ring processes is selected, using eq. (11) and fixing the maximum allowed efficiency loss; for our cluster configurations, 35 processes allow in most case to have $\gamma \leq 0.1$; then the optimal mapping of ring processes to nodes is found [5] and processes are launched.

3) The spawn of the requested number of processes is carried out in PVM using the pvm_spawn routine. In MPICH2 similar routines are available (MPI_comm_spawn and MPI_comm_spawn_multiple), but unfortunately they do not work well with our current release of MPICH2 for Windows. So, for both MPICH and MPICH2 environments, we use a procedure which spawns the optimal number of ring processes by means of $mpirun$ (MPICH) and $mpiexec$ (MPICH2) commands.

This approach assures us that the mapping is optimal, provided the load characteristics of nodes remain nearly constant during computation.

5.3. Analysis of results

The results of the trials are reported in Tables 3,4,5.

An accurate analysis of performance is difficult since various effects interact: for example the variation of processor speeds with the amount of local data, the time spent by the various processes in paging (this time varies with the amount of data of processes), the time for context switching of processes, the actual implementation of inter- and intra-node communications, etc. These effects give in some cases a superlinear speed-up.

However the main results can be summarized as follows:

a) the naive porting (one process per node) gives poor efficiencies whereas the virtual processes approach performs very well (Table 3);

b) as the degree of heterogeneity increases a higher q value is needed, on average, to achieve the same balancing (expressed by $1 - \gamma$ in Table 4);

c) all three message passing environments are able to give good performance; differences are appreciable only when the computation to communication ratio is small and in such situations MPI (both

Table 3
Execution time (in seconds) and measured (η) and simulated (η_s) efficiencies obtained on configuration (a-g) ($S = 14.24$, $h = 0.51$) using MPICH2

N	i ($\epsilon = 2^{-i}$)	T_{seq}	q	T	η	η_s
42 000	0	596.8	7	100.8	0.42	0.46
”	”	”	35	49.1	0.85	0.84
210 000	3	2614	7	316.2	0.58	0.53
”	”	”	35	158.2	1.16	0.96
420 000	3	9554	7	1366	0.49	0.48
”	”	”	35	636.5	1.05	0.88
420 000	6	975.5	7	139.7	0.49	0.49
”	”	”	35	61.2	1.12	0.89
420 000	9	191.9	7	29.5	0.46	0.44
”	”	”	35	17.7	0.76	0.77
840 000	9	751.7	7	117.8	0.45	0.46
”	”	”	35	62.8	0.84	0.82

Table 4
Comparison of three different configurations of 4 nodes with MPICH2 ($N = 105\,000$, $\epsilon = 2^{-3}$, $T_{seq} = 560.6$)

config	S	h	q	$1-\gamma$	T	η	η_s
a,b,c,d	5.30	0.25	15	0.92	112.1	0.94	0.92
a,c,e,g	7.96	0.49	25	0.94	66.3	1.06	0.94
a,e,f,g	9.94	0.60	35	0.96	57.3	0.99	0.96

implementations) outperforms PVM, owing to faster communications (Table 5);
d) MPICH2 is quite slower in the start-up phase (spawn of processes and sending of portions of points to processes);
e) the context switching overhead is always negligible for the typical number of processes per node we consider;
f) experimental and simulated results are in most cases in very good agreement (Tables 3,4,5); the most relevant source of error is probably the fluctuation of speeds with the size of local data.

6. Conclusions

We analyze and model performance of ring algorithms on heterogeneous Windows-based clusters with various message passing environments, with the aim of developing efficient and auto-adaptive applications. The general problem is exemplified considering the computation of long- and short-range interactions.

The algorithm analysis and the experimental results show that the virtual ring approach, with a number of processes much greater than the number of nodes, and with a number of processes in each node matching the node speed, is a feasible technique to exploit a very high fraction of the available power, even for platforms with a high heterogeneity. Depending on ϵ, we are able to process up to millions of high dimensional points ($m = 10$ or greater).

The algorithm is tested on a cluster of PCs connected by switched Fast-Ethernet with Windows

Table 5
Comparison of the three message passing environments (conf=(a,b,e,f,g), $S = 11.14$, $h = 0.55$, $N = 840\,000$, $\epsilon = 2^{-9}$, $T_{seq} = 751.7$)

	MPICH2			MPICH			PVM		
q	T	η	η_s	T	η	η_s	T	η	η_s
5	168.0	0.40	0.38	169.1	0.40	0.38	203.4	0.33	0.38
15	97.2	0.69	0.75	97.3	0.69	0.75	126.5	0.53	0.72
21	86.4	0.78	0.80	83.9	0.80	0.80	108.7	0.62	0.75
25	84.0	0.80	0.78	83.8	0.81	0.78	106.4	0.63	0.71
35	94.9	0.71	0.79	78.2	0.86	0.80	106.3	0.63	0.69

O.S., using three message passing libraries: MPICH2, MPICH and PVM. Whereas some differences in performance depend on the details of the implementation of the message passing library, it turns out that virtual ring algorithms for heterogeneous Windows-based clusters perform in all situations in a very satisfactory way.

The proposed analysis and procedure are quite general, and apply to any ring algorithm on heterogeneous platforms. Detailed models, as the one presented in Sect. 4, which can be very accurate, but rely on parameters which are often known in an approximate way, can be useful to understand in depth the behaviour of the application. However, in practical situations we need robust and auto-adaptive procedures, able to grasp the basic aspects of the computing platform/application.

As future work we will consider the problems arising when the load of nodes varies during computation [11,12]. In this case we need the dynamic migration of processes among the various nodes, according to node load. Another possible approach is to use threads instead of processes.

References

[1] J.K. Hollingsworth, E. Guven, C. Akinlar, Benchmarking a network of PCs running parallel applications, Proc. IEEE Int. Performance, Computing and Communications Conference, IEEE (1998) 1-7.

[2] G.C. Fox, M.A. Johnson, G.A. Lyzenga, S.W. Otto, J.K. Salmon, D.W. Walker, Solving Problems on Concurrent Processors, Vol. 1. Prentice-Hall, Englewood Cliffs, NJ (1988).

[3] A. Clematis, A. Corana, Porting regular applications on heterogeneous workstation networks: performance analysis and modeling, Parallel Algorithms and Applications 17 (2002) 205-226.

[4] A. Corana, Parallel computation of the correlation dimension from a time series, Parallel Computing 25 (1999) 639-666.

[5] P. Boulet, J. Dongarra, Y. Robert, F. Vivien, Static tiling for heterogeneous computing platforms, Parallel Computing 25 (1999) 547-568.

[6] MPICH. http://www-unix.mcs.anl.gov/mpi/mpich/

[7] MPICH2. http://www-unix.mcs.anl.gov/mpi/mpich2/

[8] PVM. http://www.csm.ornl.gov/pvm/pvm_home.html

[9] M. Fischer, J. Dongarra, Experiences with Windows 95/NT as a cluster computing platform for parallel computing, J. Parallel and Distributed Computing Practices 2 (1999).

[10] S.S. Vadhiyar, J.J. Dongarra, Self Adaptivity in Grid Computing, Concurrency and Computation: Practice and Experience 17 (2005) 235-257.

[11] O. Sievert, H. Casanova, A simple MPI process swapping architecture for iterative applications, Int. J. High Performance Computer Applications 18 (2004) 341-352.

[12] H. Dail, F. Berman, H. Casanova, A decoupled scheduling approach for Grid application development environments, J. Parallel Distrib. Comput. 63 (2003) 505-524.

Phase-Based Parallel Performance Profiling

Allen D. Malony[a], Sameer S. Shende[a], Alan Morris[a]

[a] Department of Computer and Information Science University of Oregon, Eugene, OR, USA

Parallel scientific applications are designed based on structural, logical, and numerical models of computation and correctness. When studying the performance of these applications, especially on large-scale parallel systems, there is a strong preference among developers to view performance information with respect to their "mental model" of the application, formed from the model semantics used in the program. If the developer can relate performance data measured during execution to what they know about the application, more effective program optimization may be achieved. This paper considers the concept of "phases" and its support in parallel performance measurement and analysis as a means to bridge the gap between high-level application semantics and low-level performance data. In particular, this problem is studied in the context of parallel performance profiling. The implementation of phase-based parallel profiling in the TAU parallel performance system is described and demonstrated for the NAS parallel benchmarks and MFIX application.

1. Introduction

When studying the performance of scientific applications, especially on large-scale parallel systems, there is a strong preference among developers to view performance information with respect to their "mental model" of the application, formed from the structural, logical, and numerical models used in the program. If the developer can relate performance data measured during execution to what they know about the application, more effective program optimization might be achieved. The general problem is one of *performance mapping* [13] – associating measured performance data to higher level, semantic representations of model significance to the user. The difficulty of making the association depends on what information is accessible from instrumentation and available in the measured performance data that can be used in performance mapping. There is the additional difficulty of how the performance information is to be presented to the users in model-based views.

What can be done to support the association between model and data in performance mapping? The concept of "phases" is common in scientific applications, both in terms of how developers think about the structural, logical, and numerical aspects of a computation, and how performance can be interpreted [16]. It is therefore worthwhile to consider whether support for phases in performance measurement can help to bridge the semantic gap in parallel performance mapping. Performance tracing has long demonstrated the benefits of phase-based analysis and interpretation [5]. However, performance tracing carries a heavy cost in trace size and complexity of post-mortem processing. The questions we pose in this paper concern support for phases in parallel performance profiling.

Section §2 describes the problem of phase-based profiling in more detail and outlines our objectives. The design and development approach we used to implement phase-based profiling in the TAU performance system are described in Section §3. The API for creating and controlling phases during profile measurements is also presented. We tested phase profiling on several of the NAS parallel benchmarks [1] and the MFIX (Multiphase Flow with Interphase eXchanges) [14,15] application. Section §4 shows how these programs were instrumented for phases and contrasts the phase profile results with those of standard profiling. A comparison of our work to other profiling efforts is given in Section §5. Conclusions and future work are given in Section §6.

2. Problem Description

An empirical parallel performance methodology collects performance data for *events* that occur during program execution. Thus, the events of interest determine what performance information is present in profiles or traces. Events have associated *semantics* and *context*. Semantics defines what the event represents (e.g., the entry to a routine), and context captures properties of the state in which the event occurred (e.g., the routine's calling parent). Most profiling tools can generate a *flat profile* showing how the computation performance is distributed over the program's static structure (i.e., its code points). The events usually represent routine entry and exit points, but can identify any code location. The "context" in flat profiling is the whole program. Unfortunately, a flat profile cannot differentiate performance with respect to computation dynamics.

Callgraph profiles are commonly used to separate performance with respect to parent-child event relationships. Computation dynamics is captured in the profile by identifying these relationships at the time of event occurrence. This "context" is used to essentially extend the event semantics and map performance to parents and children. Unfortunately, because callgraph profiling only captures local parent/child context, it will be difficult to map performance of deeply nested events to outermost, top-level parents. If a profiling tool supports full *callpath profiling*, the performance mapping improves because more ancestors are exposed on the calling path, but this still relies on "calling relationships" (i.e., calling contexts) to extend event semantics in the performance profile.

Given a set of instrumented events and a profile measurement infrastructure, we would like to be able to interrogate the context with each event occurrence as a way to decide how to map performance information for that event. Defining context based solely on event calling relationships is problematic because it requires events to be created for purposes of identifying context, whether or not they are of performance interest. Context relates to the notion of *state*, and its definition is richer than something that just occurs during execution (i.e., an event). Context can be logical as well as based on runtime properties. In this paper, we propose to define context in relation to the concept of a *phase*. Our goal is to then support phases as part of the instrumentation and measurement environment the developer uses to understand the performance of their application. The support should allow for phases to be created and events to be associated with phases, thereby distinguishing performance by phase identity. It should be possible to capture both *static* and *dynamic phases* in the performance measurement, and our implementation does so.

The primary contribution of our work is the realization of phase-based measurement in parallel performance profiling. Because the processing takes place online, phase support is harder to implement in profiling. Indeed, we believe our results are the first to report phase-based profiling in an actual, portable parallel performance system, TAU [9]. The following describes this implementation. We first describe the instrumentation API and how the measurement system operates to map event performance to associated phases. Examples of phase profiling with the NAS parallel benchmarks and the MFIX application are then given.

3. Implementation Approach

To understand the phase profiling approach, it is important to first discuss how flat and callpath profiling are implemented.

3.1. Flat Profiles

A flat profile shows performance data associated with different parts on an application code, typically the program routines. Profile instrumentation is placed at code points to mark the entry and

exit of execution segments and profile measurements (*inclusive* and *exclusive*) are made when these code execution events are encountered. Profile measurements are kept in a data structure TAU calls a *profile object* associated with an entry/exit event pair. A profile object can be created statically, in the sense that it will be reused, or dynamically, where a new instance will be created. A *name* is assigned to the profile object. The profile object names represent the "flat" context (i.e., it identifies only the code segement profiled). To deal with nesting, however, a flat profiling system must maintain some sort of an *event stack* (or *callstack*). However, flat profiles do not expose inter-event relationships, as no details of the program's calling order is preserved in the profiles.

3.2. Callpath Profiles

In contrast to flat profiles, callpath profiles show calling or nesting relationships between events (code segments), typically thought of as calling paths in a program's routine callgraph. In callpath profiling, events are instrumented in the same way as for flat profiling, but profile objects are created and maintained based on the context in which the events occurs (i.e., their calling or nesting context). Profile measurements are made with respect to the current callpath, which can be determined by querying the callstack on exit events. TAU encodes the callpath in the profile object's name. The *length* of a callpath is equal to the number of nested events represented in its name.

In TAU, a callpath is constructed by traversing the callstack, which exists as linked profile objects. Upon an event entry, a callpath is constructed looked up in a callpath map. If the callpath has executed before, the associated profile object is found and linked into the callstack. Otherwise, TAU creates a new profile object, initializes it with the callpath name, links it into the callstack, and adds the name to a list of events. To make the map lookup efficient, we construct its key as an array of $n + 1$ elements, where n is the length of the callpath. The top element in the array is an integer that represents the length. The next n elements are the profile object identifiers. TAU's comparison operator compares the length and the n elements successively and returns a boolean value at the first mismatch. Users can control the length of the callpath profiling to generate more a detailed or refined ancestoral view.

3.3. Phase-Based Profiles

Callpath profiling associates context with callstack state as determined by the instrumented events. However, a phase is an execution abstraction that cannot be necessarily identified by a callpath or determined by specific instrumented events. It generally represents logical and runtime aspects of the computation, which are different from code points or code event relationships. The intent of phase profiling is to attribute measured performance to the phases specified. That is, for all phases in the execution, show the performance for that phase. There are two issues to address: 1) how to inform the measurement system about a phase, and 2) how to collect the performance data.

TAU implements a phase profiling API that gives the user control to create phases and effect their transitions. The following shows the interface for C/C++ programs (TAU also supports phase profiling for Fortran):

```
TAU_PHASE_CREATE_STATIC(var, name, type, group)
TAU_PHASE_CREATE_DYNAMIC(var, name, type, group)
TAU_GLOBAL_PHASE(var, name, type, group)
TAU_GLOBAL_PHASE_EXTERNAL(var)

TAU_PHASE_START(var)
TAU_PHASE_STOP(var)
TAU_GLOBAL_PHASE_START(var)
TAU_GLOBAL_PHASE_STOP(var)
```

A phase object is constructed with a unique name when a phase is created. A phase object is similar in function to a profile object for an event. Profiling for a particular phase is started and stopped using calls that take the phase object handle. Like events (i.e., profile objects), a phase can be created as static, where the name registration and phase object construction takes place exactly once, or dynamic, where it is created and instantiated each time. Phases may be nested, in which case the "active" phase object follows scoping rules and is identified with the closest parent phase. Phases may not overlap, as exclusive time for an overlapping phase is not defined, just as it is not defined for overlapping events (profile objects). Each thread of execution in a parallel application has a default phase corresponding to the top level event, usually the main entry point of the program. This top level phase (like all phases) contains the performance of other routines and phases that it directly invokes but excludes routines called by child phases.

A phase-based profile shows the performance profile of the execution for each phase of an application, as determined by the instrumented events encountered. The user uses the phase profiling API to create, start, and stop phases. The implementation of phase-based profiling in TAU effectively will show the user a callpath performance profile of depth 2 for each uniquely identified phase. When TAU is configured with support for tracking phase profiles, the phases are activated and TAU must keep track of mappings between the instrumented events and the phase that they belong to. By default, phases are mapped by TAU to normal profile events when phase-based profiling is disabled, so as to incur no additional overhead.

The real innovation comes in the how performance data is collected for phases. To implement phase-based profiling in TAU, we leveraged our work in mapping performance data [13] and used callpath profiling (as described above) as the underlying mechanism. Here, a caller-callee relationship is used to represent phase interactions. At a phase or event entry point, we traverse the callstack until a phase is encountered. Since the top level event is treated as the default application phase, each event invocation occurs within some phase. To store the performance data for a given event invocation (in a profile object), we need to determine if the current $(event, phase)$ tuple has executed before. To do this, we construct a key array that includes the identities of the current (event) profile object and the parent phase. This key is used in a lookup operation on a global map of all phase, event relationships. If the key is not found, a new profile object is created with the name that represents the parent phase and the currently executing event or phase. In this object, we store performance data relevant to the phase. If we find the key, we access the (already existing) profile object and update its performance metrics. As with callpath profiling, a reference to this object is stored to avoid a second lookup at the event exit.

4. Application

To illustrate the use of phase-based profiling, we experimented with the NAS parallel benchmarks and other parallel applications. Our goal was to show how phase profiling can provide more refined profile results specific to phase localities than standard flat or callpath profiling. Our choice of phases and phase instrumentation boundaries was informed only by what we could learn from the application documentation and code analysis. Developers could better apply their understanding of the computational models used in the code to define phases.

4.1. NAS Parallel Benchmarks

The NAS parallel benchmark applications [1] presented convenient testcases for phase profiling. We instrumented BT, CG, LU, and SP with phases. We present only our results from BT phase profiling. The BT benchmark emulates a CFD application that solve systems of equations resulting from

an approximately factored implicit finite-difference discretization of the Navier-Stokes equations. It solves three sets of uncoupled systems of equations: first in the X, then in the Y, and finally in the Z direction. The system is block tridiagonal (hence the name) with 5x5 blocks. A multi-partition approach is used to solve the systems, since it provides good load balance and uses coarse-grained communication. BT requires a square number of processors to be used.

A natural phase analysis of the BT code would highlight performance for each solution direction. These directions are identified in the code by the three main functions `x_solve`, `y_solve`, and `z_solve`. We used static phases to see how performance varied with each direction by encapsulating the calls to these functions with phase instrumentation. Figure 1 shows the NAS Parallel Benchmark program BT run on 9 processors. The parallel system used in this case was a 16-processor SGI Altix. The top window of the figure shows a flat profile. The middle window shows the exclusive execution profile of the top-most phase, `MPBT`. It consists of the three phases (each shown separately in the bottom windows) as well as any functions called outside of any other phase, primarily `MPI_Waitall`. The `MPI_Wait` call is highlighted in all of the bottom windows enabling this routine's performance to be compared across each of the phases. We can see that `MPI_Wait` consumes different portions of the runtime between nodes in one phase, and these times are distributed differently across nodes in another phase. The portions and distributions cannot be distinguished in the

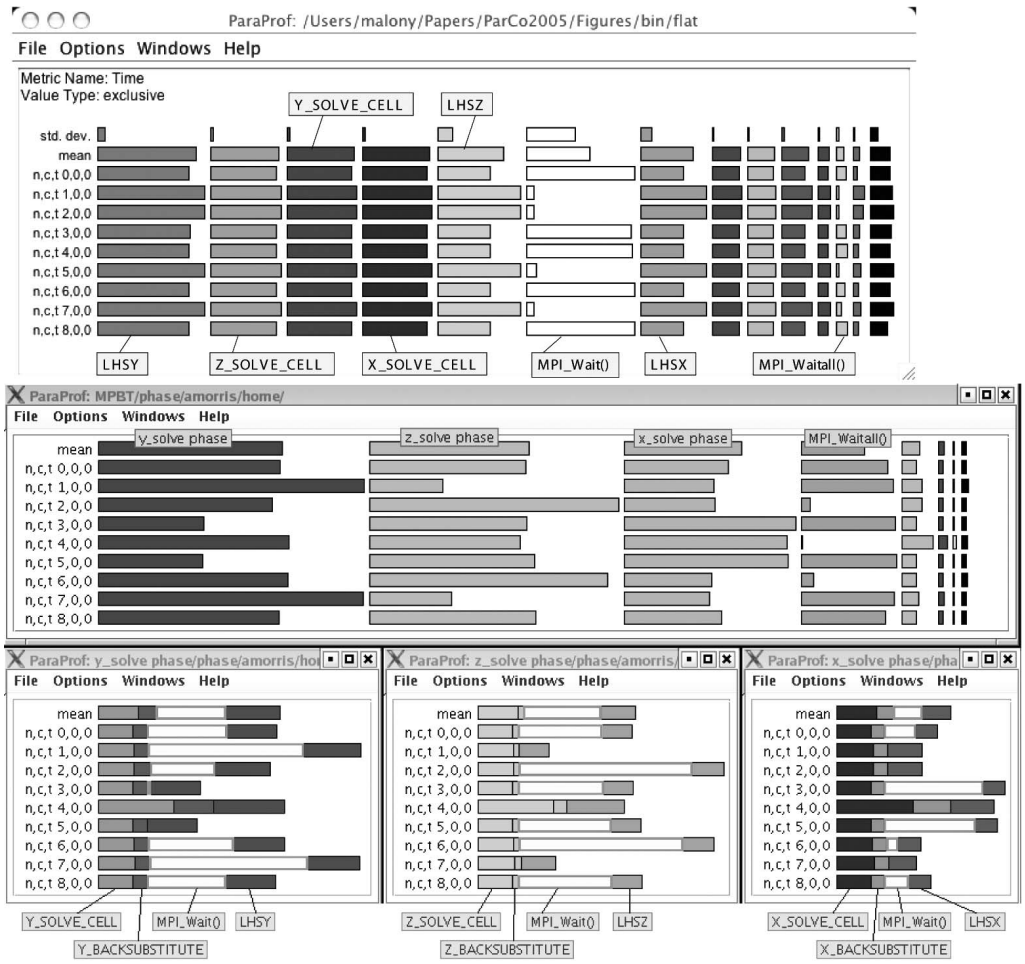


Figure 1. BT Phase Profile showing phases for x, y, and z solution directions.

flat profile.

4.2. MFIX

From the experience with the NAS benchmarks, one may conclude that callpath profiling can generate that same information as might be obtained from phase profiling. This is true *only* if the *full* callpath profile is obtained. For programs with a relatively small number of nodes in the callgraph, as is the case for BT above, a full callpath profile may be preferred. However, phase profiling can be more intuitive and closely matched to complex applications based on computational models with natural phase design and behavior, and overall less expensive than a full callpath profile. The Multiphase Flow with Interphase exchanges (MFIX [14,15]) application is such a case.

MFIX is being developed at the National Energy Transfer Laboratory (NETL) and is used to study hydrodynamics, heat transfer, and chemical reactions in fluid-solid systems. It is characteristic of large-scale iterative simulations where a major outside loop is performed as the simulation advances in time. We initially ran MFIX on a simulation testcase modeling the Ozone decomposition in a bubbling fluidized bed [4]. The flat profile for this simulation showed that over 28% of the total wallclock time was spent in the `MPI_Waitall` routine. This accounted for 70 minutes of wallclock time spent waiting for interprocess communication requests. There were 93 iterations performed by the outer simulation loop in MFIX. We were interested in knowing if all iterations took the same amount of time and how that time was distributed among the instrumented events encountered within each iteration, especially how `MPI_Waitall` varied from iteration to iteration.

The MFIX code isolated iterate computation in the Fortran subroutine `ITERATE`, making it convenient for use to instrument iterations with dynamic phases. For experimentation purposes, we ran MFIX on four processors. We then analyzed the profiles to partition the total time spent in the `MPI_Waitall` routine based on the application phase. Figure 2 shows the inclusive time contributed by each iteration. Clearly, the times spent in each iteration and in `MPI_Waitall` are not uniform. Seeing the performance profiles with respect to phases in this way provides valuable information in identifying the causes for poor performance as it relates to application specific parameters. `MPI_Waitall` acts an indicator of poor performance which can be explored in more detail by delving into the rest of the profile for the suspect phase. Figure 2 also shows the inclusive time for `SOLVE_LIN_EQ` which displays the same performance shape profile.

As this example demonstrates, phase profiling provides performance information that has previously been available only in tracing. Even if there were no difficulties generating large trace files, standard tracing tools provided no easy means to relate the performance of a group of invocations of a particular routine (such as `MPI_Waitall`, which executes over a million times on each processor in MFIX) to the application phases by simply looking at a global timeline display. The phase profiles quickly highlight the variability in performance of the `ITERATE` subroutine, which led the developers to a better understanding of the convergence criterion used for solving the set of linear equations iteratively at each time step of the computation. The performance of several routines executed within a phase are affected by convergence rate within a phase (e.g., `MPI_Waitall` and `SOLVE_LIN_EQ`). Phase-based profiling helped to track temporal differences in performance with respect to the computational abstraction the scientists understand, enabling them to be more productive in their performance problem solving.

5. Related Work

Early interest in interpreting parallel performance in relation to phases of execution grew out of two research directions. First, there was the recognition that visual patterns of performance data can

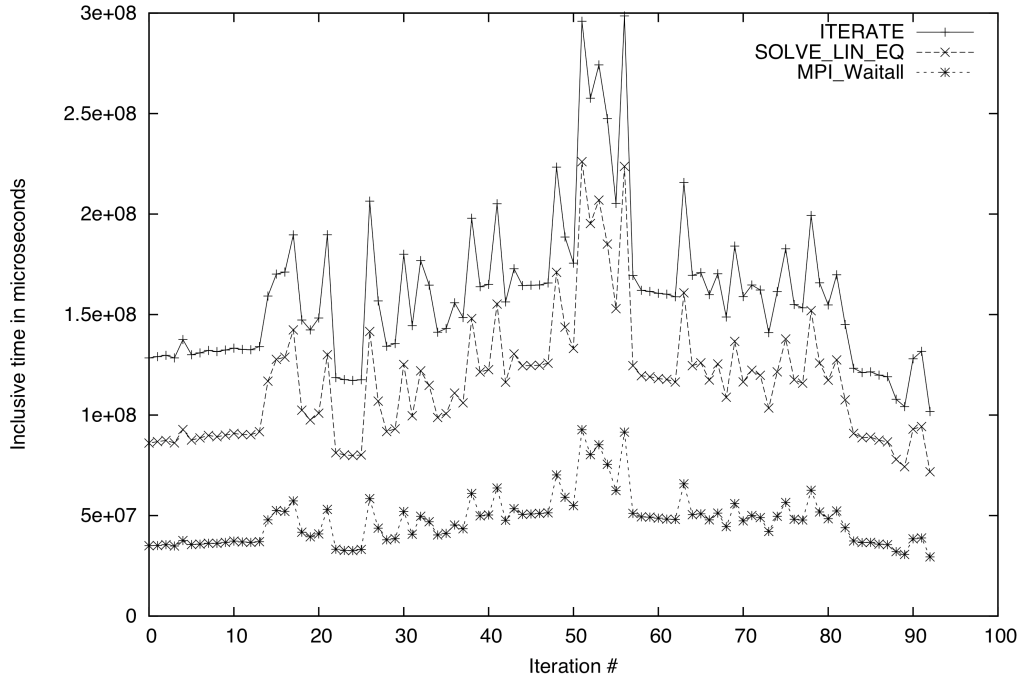


Figure 2. `MPI_Waitall` and `SOLVE_LIN_EQ` performance in each iteration

be related to application behavior [5,6,8]. The Paragraph [5] tool was the first to demonstrate how phase patterns can be seen in the visualization of performance traces. Other tools sought to represent low-level trace data in more abtract views [8] tied to application semantics. These techniques carry on in modern day trace visualization tools such as Vampir [3].

The second direction was in parallel performance modeling of scientific application codes [16]. Natural models represent the computational components and the parameters affecting their performance, as well as the sequence (phases) of their use. Phase-based performance views made it possible to relate both implementation and performance constraints between phases, versus ones that just evaluated component kernels in isolation. The approach also helped to identify performance artifacts of phase transitions, such as extra data copying or communication [2].

Irvin and Miller [7] introduced the concept of mapping low-level performance data back to source-level constructs for high-level parallel languages. They created the *Noun-Verb* (NV) model to describe the mapping from one level of abstraction to another. A *noun* is any program entity and a *verb* represents an action performed on a noun. Sentences, composed of nouns and verbs, at one level of abstraction, map to sentences at higher levels of abstraction. The *ParaMap* methodology defined three different types of mappings: static, dynamic, and one based on the *set of active sentences* (SAS). Shende's [13] *Semantic Entities, Associations, and Attributes* (SEAA) model builds upon the NV/SAS mapping abstractions to support user-level mapping, more accurate performance capture, and asynchronous operation. Our phase-based profiling research is founded in these mapping principles.

6. Conclusion

Linking performance data to execution semantics requires mapping support in the measurement system. We have developed techniques to map performance profiles to phases of a computation.

Phases are defined by the application developer, using an API for phase creation and control. Phases can be used to represent structural, logical, and execution time aspects of the program. Phases can also be hierarchically related. Parallel profiles produced at the end of an execution encode phase relationships. For any particular phase, its profile data reflects the parallel performance when that phase was active in the computation.

We have implemented phase-based profiling in the TAU parallel performance system and this paper demonstrates the implementation for the NAS benchmarks and the MFIX application. In addition, we have applied phase profiling to several large-scale applications including a high-fidelity simulation of turbulent combustion with detailed chemistry (the S3D code [11]), a simulation of astrophysical thermonuclear flashes (the FLASH code [10]), and the Uintah computational framework for the study of computational fluid dynamics [12]. Our future objectives are to improve the phase profile data analysis and provide to better visualization of phase performance.

References

[1] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, M. Yarrow, "The NAS Parallel Benchmarks 2.0," Technical Report NAS-95-020, NASA Ames Research Center, 1995.

[2] J. Brehm, P. Worley, and M. Madhukar, "Performance Modeling for SPMD Message-Passing Programs," *Concurrency: Practice and Experience*, **10**(5):333–357, April 1998.

[3] H. Brunst, M. Winkler, W. Nagel, H.-C. Hoppe, "Performance Optimization for Large Scale Computing: The Scalable VAMPIR Approach," In V. Alexandrov, J. Dongarra, B. Juliano, R. Renner, K. Tan, (eds.), *International Conference on Computational Science*, Part II, LNCS 2074, Springer, pp. 751–760, 2001.

[4] C. Fryer and O.E. Potter, "Experimental Investigation of models for fluidized bed catalytic reactors," AIChE J., **22**:38–47, 1976.

[5] M. Heath and J. Etheridge, "Visualizing the Performance of Parallel Programs," *IEEE Software*, **8**(5):29–39, September 1991.

[6] M. Heath, A. Malony, and D. Rover, "The Visual Display of Parallel Performance Data," *IEEE Computer*, pp. 21–28, November 1995.

[7] R. Irvin and B. Miller, "Mapping Performance Data for High-Level and Data Views of Parallel Program Performance," *International Conference on Supercomputing (ICS)*, May 1996.

[8] T. LeBlanc, J. Mellor-Crummey, and R. Fowler, "Analyzing Parallel Program Executions using Multiple Views," *Journal Parallel and Distributed Computing*, **9**(2):203–217, June 1990.

[9] A. Malony, S. Shende, R. Bell, K. Li, L. Li, and N. Trebon, "Advances in the TAU Performance System," in *Performance Analysis and Grid Computing*, (Eds. V. Getov, et. al.), Kluwer, Norwell, MA, pp. 129-144, 2003.

[10] R. Rosner et al., "Flash Code: Studying Astrophysical Thermonuclear Flashes," *Computing in Science and Engineering*, **2**:33, 2000.

[11] http://scidac.psc.edu/

[12] S. Shende, A. Malony, A. Morris, S. Parker, J. de St. Germain, "Performance Evaluation of Adaptive Scientific Applications using TAU," *International Conference on Parallel Computational Fluid Dynamics*, 2005.

[13] S. Shende, "The Role of Instrumentation and Mapping in Performance Measurement," Ph.D. Dissertation, University of Oregon, 2001.

[14] M. Syamlal, W. Rogers, and T. O'Brien, "MFIX documentation: Theory Guide," Technical Note, DOE/METC-95/1013, 1993.

[15] M. Syamlal, "MFIX documentation: Numerical Technique," EG&G Technical Report, DE-AC21-95MC31346, 1998.

[16] P. Worley, "Phase Modeling of a Parallel Scientific Code," *Scalable High-Performance Computing Conference (SHPCC)*, pp. 322–327, 1992.

Tracing the Cache Behavior of Data Structures in Fortran Applications

Laszlo Barabas[a], Ralph Müller-Pfefferkorn[a], Wolfgang E. Nagel[a], Reinhard Neumann[a]

[a]Center for Information Services and High Performance Computing (ZIH)
Dresden University of Technology
D-01062 Dresden, Germany
{barabas,mueller-pfefferkorn,nagel,neumann}@zhr.tu-dresden.de

In an application, data access can become a major performance bottleneck if the memory hierarchy of the underlying hardware architecture is not taken into account. The only way to gain deeper insight of an applications memory usage is to measure its data access behavior with hardware counters. From the programmer's point of view such performance data (like cache misses or hits) have to be linked to the data structure causing it. The name of a data structure is the only point of reference the user has and the only point where he can apply optimizations.

In the project EP-Cache tools for Fortran applications were developed to monitor and to link hardware counter information with data structures. This includes an appropriate visualization of the gathered information. Parallelism using the OpenMP programming paradigm is also supported.

1. Introduction

"Processor performance is not an issue." At least in the last years and, probably, up to 2012 this statement was and is still valid. Following Moore's law the speed and capabilities of modern microprocessors have increased significantly in the past and will do so in the near future.

Nevertheless, the overall performance of a computer depends on more than pure CPU power. Memory access is (still) a bottleneck. Memory performance increases only by about 7% per year. The concept of hierarchical cache levels providing faster access to parts of data improves the situation but needs a strict adoption of programming to its paradigms. Compilers and other optimization tools might support programmers to reach this goal. But this job is very complex and there are many influencing factors like data layout, details of the hardware structure etc. Only in a multistage development process of "coding - testing - monitoring" can an optimum be achieved.

In applications for numerical calculations or for data processing large data arrays, vectors or other structures are usually used. The employment of data access techniques which exactly reflect the memory hierarchy of caches and main memory of the underlying hardware system are crucial to gain the best performance in such cases. The "only" problem is to know, how a code makes use of the memory hierarchy. This is a nontrivial problem, as in most cases software is too complex for theoretical consideration to reveal the true nature of the data access at the instruction level. The only way to find that out is to measure the system's data access behavior.

Hardware counters that measure cache and memory hits and misses are a well known mechanism to gather data access information. Many tools were written to collect these information

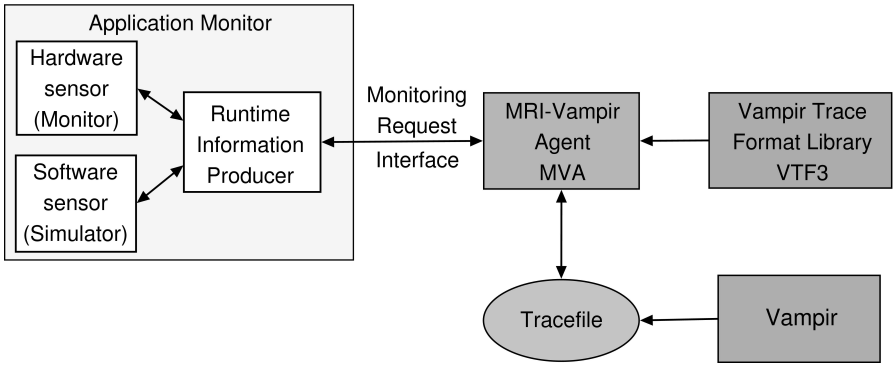


Figure 1. Architecture of the hardware and simulation based monitoring system

as profiling data, giving a summary about an applications behavior. More sophisticated tools even provide counter data for regions of a code allowing a more structural view on its data accesses. But to draw a direct line between a single data structure in a code region and a counter information is a problem only few profiling tools can solve [5].

For years, performance monitoring of programs was successfully done by writing out detailed information to trace files during program execution. In a second step, and (usually) after the execution had finshed, the collected data were analyzed using visualization techniques. Due to the complexity of modern applications the visualization has become an essential part of a performance analysis and often the only way to understand in detail what was going on during the program run.

The aim of the project EP-Cache[1] [2] was to develop tools to support Fortran programmers in the monitoring and optimization steps, specializing on data structures and their cache access.

In this paper, we want to present some of the results of these efforts, focusing on the tools for monitoring data structure related cache performance and their visualization. In the sections 2 and 3 we want to describe the two approaches to collect data structure related performance data. Section 4 reports about the extensions needed for the tracing infrastructure that was used to record the performance data. Finally, section 5 describes the new visualizations that were implemented into *Vampir* [3], the well-known performance analysis suite, to effectively analyse the collected data.

2. The Hardware Monitor EPCMON

At the TU München, a hardware monitoring environment was developed to expand the classical hardware counter concept and to allow the measurement of data structure related information at runtime.

The hardware monitor can run in two modes. The *static mode* allows to count address specific events. Such events can be e.g. L1 misses for array A in one or several specific code regions. The *dynamic mode* enables a monitoring of accesses to address ranges. It delivers the data in the form of a histogram with a configurable granularity as multiples of cache lines. An example could be a histogram with 30 bins gathering L1 cache hits for an integer array C[120] in a specific code region. With a cache line size of 8 bytes and an integer length of 2

[1]funded by the German Federal Ministry of Education and Research (BMBF) under contract 01IRB04

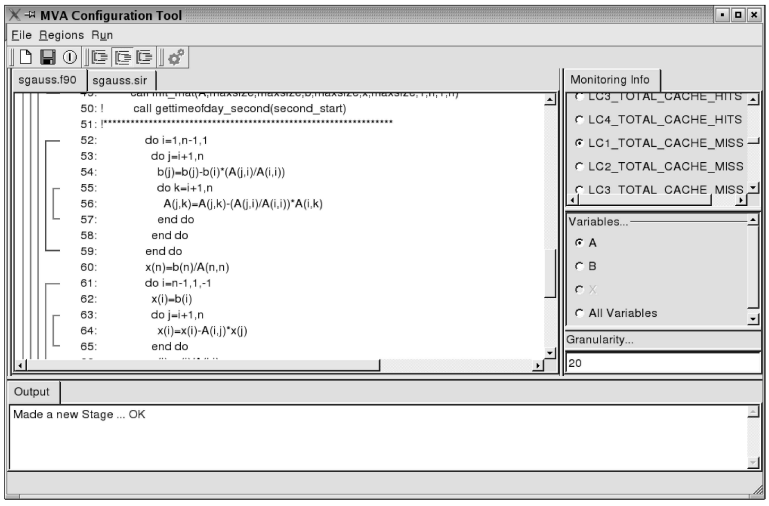


Figure 2. The graphical user interface MVAconfigure eases the specification of monitoring requests

bytes one bin would hold the values for 4 elements of array C.

As the hardware monitor is still a concept currently, a simulation of the monitor was implemented to test and verify its concepts. It is based on the Valgrind [8] runtime instrumentation framework for IA-32 architectures and allows a on-the fly cache simulation for OpenMP programs.

To define regions in the source code for the measurement of the performance data and to map the virtual addresses or address ranges to data structures instrumentation of the source code is used.

Both, the hardware monitor and the simulation can be accessed via a common interface - the *Monitoring Request Interface* (MRI) [4]. MRI allows a consumer to specify monitoring requests and to retrieve the monitor data. A request can look like: Monitor the L2 cache misses for array A in the file matmul.f90 in the region from lines 10 to 14 and write the data in a histogram with 20 bins!

A more detailed description of the hardware monitor and the simulator can be found in [6].

To steer the monitor and to write out the performace data to tracefiles (see section 4) a tool was implemented (MRI-Vampir-Agent, MVA). MVA reads in a configuration file with monitoring requests, transfers the requests to the monitor, reads out the performance information and writes them to a tracefile. Beside the new data structure related counter information (see section 4) "traditional" trace events (like enter and exit of regions or OpenMP related events) are written out. Fig. 1 illustrates the basic architectural concept of the monitor in combination with the *VTF3* tracing infrastructure and a *Vampir* based performance analysis (see sections 4 and 5).

To create the configuration file with the monitor requests a graphical user interface (MVAconfigure) was written. The user simply loads his source code into MVAconfigure and with some clicks he can select the regions, the data structures, the hardware counter and other configurations for the requests (see fig. 2).

3. Source Code Instrumentation Based Monitoring

Another approach to gather data structure related performance data does not use special hardware or simulation. It is based on the ADAPTOR compilation system ([1], SCAI FhG Sankt Augustin) and combines three techniques to get the needed information.

- source code instrumentation
- precise event-based sampling (PEBS) of hardware counters
- mapping of addresses to data structures

The original source code is instrumented automatically for tracing using ADAPTOR, compiled and run. The user can select the regions, counters and data structures for monitoring.

In a region, where the data access behavior should be observed, a precise event-based sampling is started. The technique is, that e.g for every 200th L1 cache miss an interrupt is generated. Modern processors (as the Pentium IV or the Itanium) then allow the readout of the (precise) current instruction pointer and all general purpose registers. With this information, one can identify the current instruction and compute the data addresses used in it. Such, a data address profile for specific events (like cache misses) in one or more specific regions can be collected.

The final step is to translate the data addresses to the corresponding data structure in the source code. This is done with ADAPTOR's runtime memory management. By instrumenting the source code the addresses of the data structures can be determined at runtime and a reverse mapping can be applied. Thus, dynamically allocated memory can also be included.

Currently, the monitoring runs on two platforms that support precise event-based sampling: Pentium IV and Xeon machines (Linux with the perfctr [9] and hardmeter [11] packages) and on Itanium machines (Linux with libpfm [7]).

4. The VTF3 trace infrastructure

For both the hardware monitor/simulator and the instrumentation based monitoring, the trace data is stored in a *VTF3* tracefile [10]. *VTF3* is a trace infrastructure developed at ZIH. *VTF3* can be read and visualized by *Vampir* for post-mortem performance analysis.

VTF3 provides definitions and records for various kinds of trace events, such as the definition of the computing environment (e.g. number of CPUs or CPU grouping), enter/exit of code regions, hardware counter data, MPI communication, OpenMP events etc. They are stored together with a time stamp to get a temporal view of the applications run.

In the EP-Cache project *VTF3* was extended to store records for data structure related performance data. 4 new records were defined and implemented. They allow the definition of data structures and histograms.

For data structures, the name and source code locations (file name, line numbers) can be recorded.

A histogram represents a vector of (counter) values. For each vector element (each bin of the histogram) the data structures, which the counter values were collected for, are stored. Other general information for the histogram includes timestamps (start and end time of the data collection), the kind of counter information, process information and more.

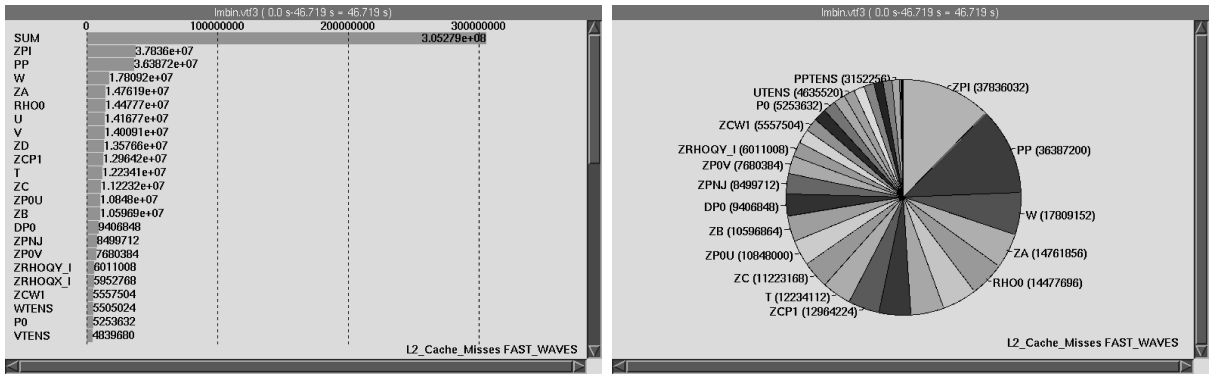


Figure 3. Global summary of the cache usage of the monitored data structures, bar and pie representation

5. Visualizing data structure related performance data

Vampir is a visualization and performance analysis environment, which has been developed at the ZIH for many years. *Vampir* is used at many HPC centers worldwide. It reads information from tracefiles and transfers them to a variety of graphical displays (e.g. timelines, state diagrams, statistics . . .), supporting users in the analysis and optimization of their programs.

To visualize the new data structure related performance data in a typical *Vampir*-like style, new displays were developed and implemented. Thus, various possibilities are available to the user to analyse performance losses resulting from the cache behavior of his application.

Depending on the users choice, the data are visualized either time dependent with as a so called "timeline" or as summaries. The new features are described in the following.

5.1. Summary data

A first glance on the overall cache performance of the data structures of a program gives the display *Cache Data Summary*. It summarizes the values of the performance counters for the monitored data structures. The information can be shown either in a bar or a pie representation (fig. 3). In the bar presentation the data structures can be sorted depending on the counter values. The pie representation gives the possibility to view the fraction each data structure contributes.

In the *Cache Bin Statistics* (fig. 4(a)) the counter values are displayed as a histogram, where each bin represents either one data structure, parts of a data structure or several data structures.

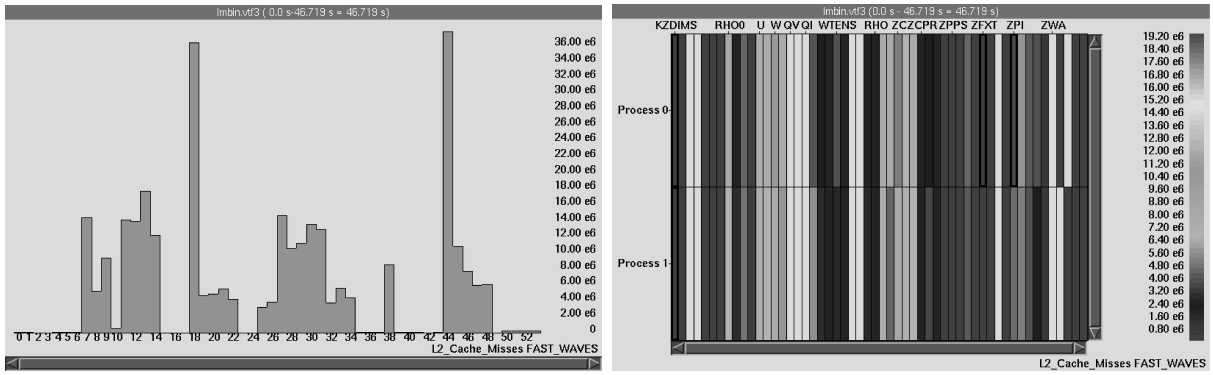


Figure 4. Cache Bin Statistics (left) and Cache Data Statistics for an example with 2 processes (right)

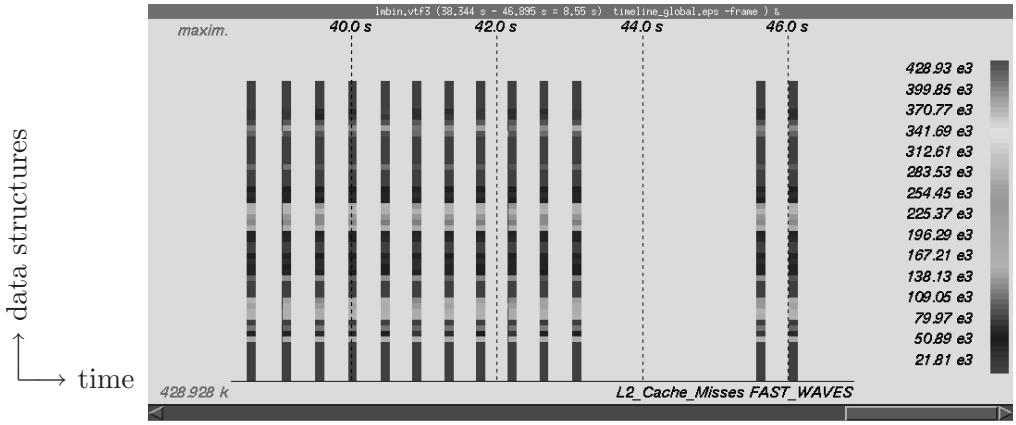


Figure 5. Global cache timeline (summarized over all processes)

The distribution of counter values to processes and data structures is shown in the *Cache Data Statistics* display. The counter values are colorcoded for each bin. Thus, a direct comparison of the cache behavior of the data structures in the individual processes is possible.

For all of the above displays the shown summaries depend on the timeframe the user selected in any timeline - zooming in or out in time, recalculates and adjusts the displays. Additionaly, individual processes can be filtered out for the calculation of the statistics.

5.2. Cache Timelines

The most interesting new feature are the Cache Timelines that visualize the temporal development of the cache behavior of the monitored data structures. The Cache Timelines show the histograms of counter values of the data structures (ordinate) as a function of time (abscissa). The time spread of the histograms is given by the start and end time of the monitoring in a region. The counter values of the individual bins are colorcoded.

In the *Colorcoded Cache Timeline* the counter values, summarized over all processes, are displayed. In the example in figure 5, which shows the L2 misses of all data structures of one subroutine (FAST_WAVES), one can clearly see, that there are some data structures with a large number of L2 cache misses and others with almost no misses. But over time, the cache behavior of the single data structures does not change significantly.

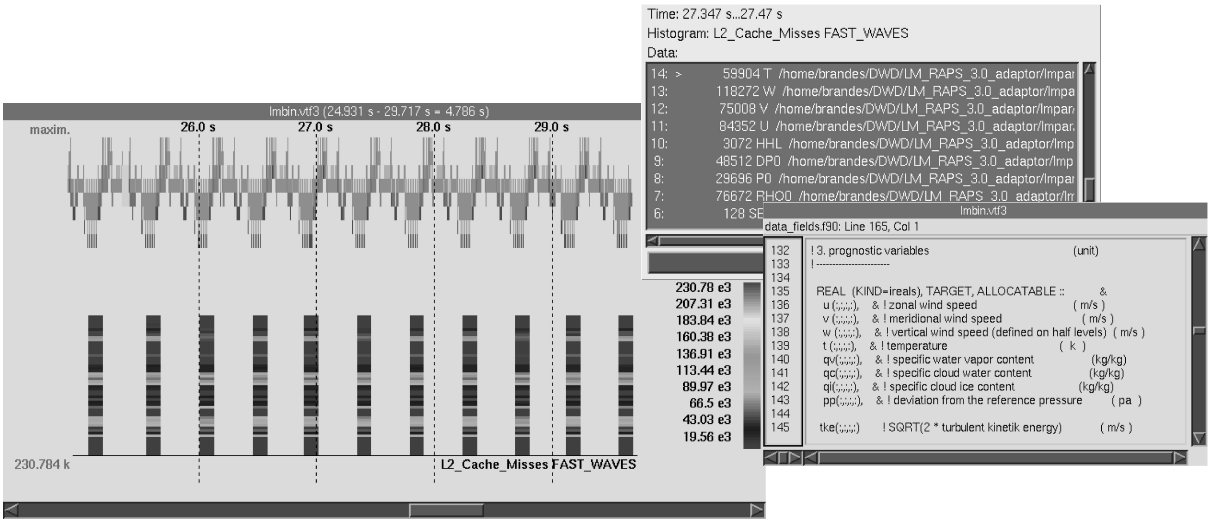


Figure 6. Process cache timeline with histogram information window and source view

If the user is interested in the data of a single process only, he can select one (in the *Vampir* timeline display) and take a look at the *Process Cache Timeline* (fig. 6). Below the call stack of the selected process the Cache Timeline is displayed. The combination of both allows the user to better analyse and understand the correlation between the call of single code regions and the cache behavior of data structures in these regions.

5.3. Connection to source code

In all of the displays described above the user has direct access to information about the data structures. Clicking e.g. on a bin in a histogram opens up a window with the name of the data structure monitored and the corresponding counter value (see fig. 6). A click on the name of the data structure brings up another window containing the relevant source code.

6. An Illustrating Example

The more complex a code is the more difficult it is to specify exactly the reason for a performance loss. Especially in numerical calculations or in simulations where complex calculations are done (often on one code line), the kind of access to a single data structure can cause a significant performance decrease.

In the following (simple, and thus comprehensible) example we want to illustrate the benefits the monitoring of data structure related performance data provides.

```
!ADDITION OF MATRIX ELEMENTS
!REGION 1
DO I=1,N
   DO J=1,N
     S=S+A(I,J)+B(J,I)
   ENDDO
ENDDO
!REGION 2
DO J=1,N
   DO I=1,N
     S=S+A(I,J)+B(I,J)
   ENDDO
ENDDO
```

In region 1, array A is accessed in the wrong order - Fortran stores arrays columnwise in memory. Array B is accessed in the correct order. In region 2 both A and B are accessed correctly in columnwise order. Monitoring the L1 cache misses without a correlation to the data structures, reveals a cache access problem in region 1 (fig. 7a) - but no sign of what is causing it. (There are still misses in region 2, because A and B do not fit into the cache.) Fig. 7b shows the colorcoded cache timeline of the L1 cache misses for array A and B separately. It clearly indicates, that the number of L1 cache misses for A in region 1 (wrong order) is much larger than the one in region 2. For array B the number of misses is the same for both regions. Thus, it shows clearly that only the access to A is the problem.

7. Summary

In the EP-Cache project a number of tools were developed that allow the monitoring of data structure related cache performance counters in Fortran applications. New extensions enable the *VTF3* trace infrastructure to record such information, e.g as histograms, for a post-mortem analysis. The performance-analysis environment *Vampir* was equipped with displays

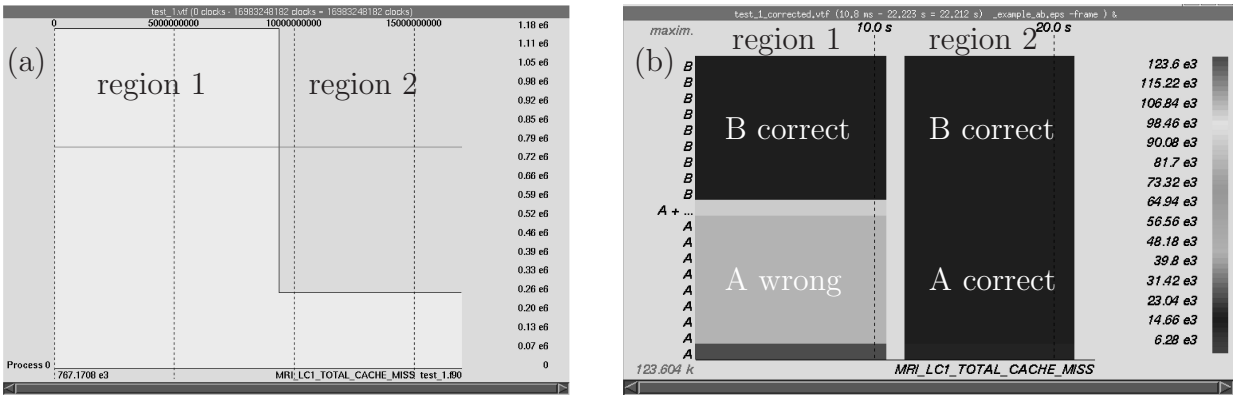


Figure 7. (a) counter timeline: L1 cache misses without correlation to data structures; (b) colorcoded cache timeline for L1 cache misses: array A - wrong access order in first, correct access order in second region; correct access order for array B in both regions

to visualize the new data structure related cache performance data. This enables users to analyse their code regarding the cache behavior of individual data structures, thus providing the posssibility to optimize the data access patterns at source code level.

References

[1] T. Brandes. *ADAPTOR - Automatic DAta Parallelism TranslaTOR.* Institute for Algorithms and Scientific Computing (SCAI FhG), Sankt Augustin, 2000. http://www.scai.fhg.de/index.php?id=291&L=1.

[2] Th. Brandes, H. Schwamborn, M. Gerndt, J. Jeitner, E. Kereku, W. Karl, M. Schulz, J. Tao, H. Brunst, W. E. Nagel, R. Neumann, R. Müller-Pfefferkorn, B. Trenkler, and H.-Ch. Hoppe. Monitoring Cache Behavior on Parallel SMP Architectures and Related Programming Tools. *Journal on Future Generation Computing Systems*, 2005.

[3] Holger Brunst, Wolfgang E. Nagel, and Allen D. Malony. A distributed performance analysis architecture for clusters. In *IEEE International Conference on Cluster Computing, Cluster 2003*, pages 73–81, Hong Kong, China, December 2003. IEEE Computer Society.

[4] M. Gerndt and E. Kereku. Monitoring request interface version 1.0. Technical report, TU München, 2004. http://wwwbode.in.tum.de/∼ kereku/epcache/pub/MRI.pdf.

[5] M. Itzkowitz, B. J. N. Wylie, Ch. Aoki, and N. Kosche. Memory Profiling using Hardware Counters. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, Washington DC, USA, 2003. IEEE Computer Society.

[6] E. Kereku, T. Li, M. Gerndt, and J. Weidendorfer. A Selective Data Structure Monitoring Environment for Fortran OpenMP Programs. In D. Laforenza M. Danelutto, M. Vanneschi, editor, *Euro-Par 2004 Parallel Processing*, page 133, Pisa, Italy, 2004. Springer-Verlag.

[7] HP Labs. *Performance Analysis Tools for the IA-64.* http://www.hpl.hp.com/research/linux/perfmon/.

[8] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In *Proceedings of the Third Workshop on Runtime Verification (RV03)*, Boulder, Colorado, USA, July 2003.

[9] M. Pettersson. *Linux x86 Performance-Monitoring Counters Driver.* http://www.csd.uu.se/ mikpe/linux/perfctr/.

[10] S. Seidl, A. Knüpfer, and R. Müller-Pfefferkorn. VTF3 - A Fast Vampir Trace File Low-Level Management Library. Technical report, ZIH TU Dresden, 2004.

[11] K. Takehiro and Hiro Yoshioka. *Hardmeter - a memory profiling tool*, 2003. http://sourceforge.jp/projects/hardmeter.

Algorithms

A Parallel Variant of the Gram-Schmidt Process with Reorthogonalization

V. Hernandez[a], J. E. Roman[a], A. Tomas[a]

[a]D. Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, Camino de Vera s/n, 46022 Valencia, Spain

1. Introduction

This work is concerned with the effective parallelization of algorithms for computing an orthonormal basis of a subspace in the context of Krylov-type eigenvalue solvers such as Arnoldi. The main motivation is to try to improve the parallel efficiency of eigensolvers implemented in SLEPc, the Scalable Library for Eigenvalue Problem Computations [4], especially when addressing large-scale eigenproblems with many processors in distributed memory platforms.

It is well known that the scalability of Krylov-type eigensolvers is limited by the synchronization points associated to vector norms and inner products, in particular those required during the orthogonalization of vectors. Typically, orthogonalization in this context is carried out by means of some variant of the Gram-Schmidt process. In this work, a new algorithm is proposed, which is a modification of classical Gram-Schmidt with reorthogonalization, that can be parallelized with fewer synchronization points thus leading to better parallel performance.

The text is organized as follows. Section 2 provides an overview of orthogonalization algorithms used in the context of eigensolvers. The proposed algorithm is described in detail in section 3. Section 4 presents timing results measured in different parallel platforms, which illustrate the benefits of the new algorithm in terms of parallel performance. Finally, a discussion is given in section 5.

2. Orthogonalization in the Context of Eigensolvers

Given a standard eigenvalue problem, $Ax = \lambda x$, or a related eigenproblem, where A is a square matrix of order n, the goal is to compute the eigenvalue, λ, or the eigenvector, x, or both.

Krylov-type eigensolvers such as Arnoldi are based on explicitly building an orthonormal basis of a Krylov subspace, $\mathcal{K}_m(A, v_1) := \mathrm{span}\{v_1, Av_1, \ldots, A^{m-1}v_1\}$, and then selecting members of that subspace as approximations of the eigenvectors. These iterative eigensolvers operate with matrix-vector products, thus preserving sparsity. That makes them especially appropriate for sparse problems such as those arising after the discretization of partial differential equations. They are also well suited for addressing huge problems in parallel computers, as discussed in subsection 2.2.

Algorithm 1 illustrates the Arnoldi method schematically, denoting by V_m the $n \times m$ matrix with column vectors $v_1, \ldots, v_m$ and by H_m the $m \times m$ Hessenberg matrix whose nonzero entries are h_{ij}.

Algorithm 1 (Arnoldi)

Input: Matrix A, number of steps m, and initial vector v_1 of norm 1
Output: $(V_m, H_m, v_{m+1}, h_{m+1,m})$ so that $AV_m - V_m H_m = h_{m+1,m} v_{m+1} e_m^T$

```
for j = 1, 2, ..., m
    Expand the basis: w_{j+1} = A v_j
    Orthogonalize w_{j+1} with respect to V_j, obtaining w''_{j+1} and coefficients h_{1:j,j}
    Normalize: h_{j+1,j} = ||w''_{j+1}||_2,    v_{j+1} = w''_{j+1} / h_{j+1,j}
end
```

It can be shown (see for example [7]) that Algorithm 1 computes an orthonormal basis V_m of the Krylov subspace $\mathcal{K}_m(A, v_1)$. The Arnoldi algorithm is numerically superior to the straightforward approach of computing all the basis vectors and then orthogonalizing them. The reason for this is that the trivial basis of the Krylov subspace, $\{v_1, Av_1, \ldots, A^{m-1}v_1\}$, is very ill-conditioned and linear independence is lost very rapidly in finite precision. Arnoldi avoids this by orthogonalizing a vector that is a candidate for expanding the basis as soon as it is generated.

In the Arnoldi scheme, since only one vector at a time has to be orthogonalized, the Gram-Schmidt orthogonalization procedures come in very naturally. Orthogonalization via Householder reflectors could also be used, as proposed in [10], but at the expense of higher computational cost and more difficult implementation. This work focuses on Gram-Schmidt procedures.

In order to guarantee the numerical stability of Algorithm 1, the vector computed in the orthogonalization step, w''_{j+1}, must be orthogonal to full working precision with respect to the columns of V_j. Indeed, if the quality of orthogonality is poor then instabilities may appear. Therefore, care must be taken so that the chosen orthogonalization algorithm meets this requirement.

Several variants of Gram-Schmidt orthogonalization exist. It is well known that Modified Gram-Schmidt (MGS) provides much better quality of orthogonality than Classical Gram-Schmidt (CGS). This is the reason why MGS is preferred in linear solvers such as GMRES. However, in the context of eigensolvers the MGS scheme is not sufficient for all cases: large cancellations can still occur and in that case the computed basis fails to be orthogonal to full working accuracy. A cure for this is to resort to double orthogonalization, as described next.

2.1. Gram-Schmidt with Reorthogonalization

The simplest possibility to guard against large cancellations in Gram-Schmidt orthogonalization is to take the resulting vector and perform a second orthogonalization. This approach is usually referred to as CGS2 and MGS2 for the classical and modified variants, respectively. Algorithm 2 shows the CGS2 algorithm, where w_{j+1} denotes the initial vector, w'_{j+1} the vector resulting from the first orthogonalization, and w''_{j+1} the vector obtained after reorthogonalization. The computed coefficients h_j are referred to as $h_{1:j,j}$ in Algorithm 1.

Algorithm 2 (Classical Gram-Schmidt with Reorthogonalization, CGS2)

Input: Vector w_{j+1} to be orthogonalized against the columns of V_j
Output: Orthogonalized vector w''_{j+1} and coefficients h_j

$h_j = V_j^T w_{j+1}$
$w'_{j+1} = w_{j+1} - V_j h_j$
$c_j = V_j^T w'_{j+1}$
$w''_{j+1} = w'_{j+1} - V_j c_j$
$h_j = h_j + c_j$

Note that, in exact arithmetic, the coefficients c_j are zero and vectors w'_{j+1} and w''_{j+1} coincide. However, this is not the case in finite precision arithmetic, where c_j can be thought of as a maybe-not-so-small correction to h_j.

In cases where large cancellations have not occurred in first place, reorthogonalization is superfluous. On the other extreme, it could happen that large cancellations occur also in the reorthogonalization stage, thus requiring yet another reorthogonalization. In general, an iterative scheme can be defined in which a simple criterion, such as the one described in [2], is used to determine whether the computed vector is good enough. In practical situations, typically only one reorthogonalization is necessary at most. A detailed description of iterative Gram-Schmidt procedures can be found in [5].

2.2. Parallel Implementation of the Arnoldi Method

Assuming a message-passing paradigm with standard data distribution of vectors and matrices, i.e. by blocks of rows, parallelization of Algorithm 1 amounts to carry out the three stages in parallel:

1. Basis expansion. In the simplest scenario, this stage consists in a sparse matrix-vector product. In most applications, this operation can be parallelized quite efficiently, provided that the amount of data to be exchanged among processors is reduced, for example, by using a nearly optimal ordering computed via graph partitioning algorithms.
2. Orthogonalization. As exemplified in Algorithm 2, this stage consists in a series of vector operations such as inner products, additions, and multiplications by a scalar.
3. Normalization. The computation of the norm requires a global reduction operation, thus representing a synchronization point in the algorithm.

Since vector addition and scaling are trivially parallelizable operations, the efficiency of the orthogonalization stage is given by the number of required inner products. More precisely, what is important is the number of synchronization points, since the data required to be exchanged for several independent inner products can be combined in a single reduction operation, resulting in almost the same communication cost as just one inner product. Therefore, the scalability of the different variants of Gram-Schmidt orthogonalization depends on the data dependency of inner products. This is the reason why the CGS versions scale better than the MGS counterparts, since inner products associated to operations such as $h_j = V_j^T w_{j+1}$ can be grouped together in a single communication.

3. The Proposed Algorithm

The new algorithm proposed to improve the parallel efficiency of the orthogonalization stage is based on Algorithm 2. The first part of this section tries to justify this decision.

3.1. Motivation and Related Work

It should be clear from the above discussion that the orthogonalization stage is mainly the source of poor scalability of Arnoldi's method. Other authors have already attempted to enhance this stage in terms of parallel efficiency, by rearranging the computations in such a way that more than one vector at a time is ready for being orthogonalized [8], or by addressing the problem in the context of block-Krylov eigensolvers [9]. In contrast, the approach presented below is still compatible with the simpler, single-vector scheme of Algorithm 1.

As stated in subsection 2.2, parallel implementations of CGS scale better than in the case of MGS. In addition, CGS usually provides higher Mflop rates even for one processor, due to a better data access pattern. On the other hand, from the numerical point of view both CGS2 and MGS2 are equivalent in the sense that reorthogonalization repairs the effect of cancellation, as shown in [5]. For all these reasons, CGS2 has been chosen as the starting point for deriving the new algorithm.

In this work, we consider CGS2 without selective reorthogonalization, that is, reorthogonalization is carried out in all cases. As mentioned in subsection 2.1, this implies a higher computational cost compared to the variant in which a criterion is used. However, this drawback is not so serious as it may seem, since it turns out that the average number of orthogonalizations in a selective reorthogonalization variant is quite often close to 2 rather than close to 1. The next table shows the average number of orthogonalizations for several test cases taken from the NEP [1] and UF [3] collections.

AF23560	1.07	ANDREWS	1.85	BWM2000	1.90	CRY10000	1.86	CDDE1	1.81
DW8192	1.60	LIN	1.88	LOP163	1.21	OLM5000	1.87	PDE2961	1.41
QH1484	1.94	RDB2048	1.80	RW5151	1.07	TOLS4000	1.99	TUB1000	1.89

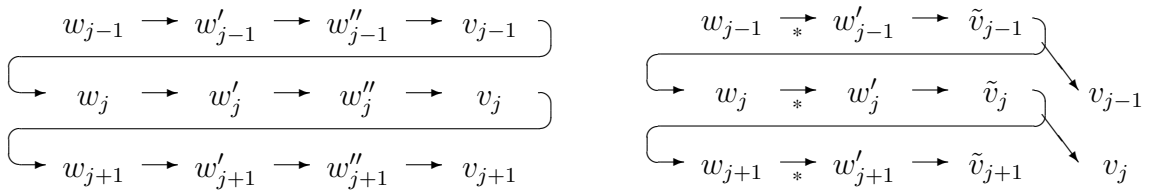


Figure 1. Computational schemes of CGS2 (left) and DCGS2 (right) algorithms.

3.2. Delayed Reorthogonalization

The key idea of the proposed algorithm is to delay the reorthogonalization (last three lines in Algorithm 2) in such a way that communication associated to it can be merged with that of the first orthogonalization of the next vector, which corresponds to the subsequent Arnoldi iteration. We refer to this algorithm as Delayed Classical Gram-Schmidt with Reorthogonalization (DCGS2).

In DCGS2, the next Arnoldi iteration is started as soon as the result of the first orthogonalization is available, w'_{j+1}, and reorthogonalization is postponed until h_{j+1} is to be computed. Figure 1 illustrates the CGS2 and DCGS2 computation schemes for several iterations. In CGS2, the sequence of steps necessary to compute v_{j+1} from v_j are followed in a conceptually linear fashion, proceeding to the next basis vector only after the current vector is completely "finished". In contrast, in the new algorithm, part of the computation associated to a vector is interleaved with the computation of the next one. Note that in DCGS2 the notation changes slightly: now the normalization is carried out over w'_{j+1}, producing $\tilde{v}_{j+1}$, which represents a vector that still requires a reorthogonalization to become the final v_{j+1}. What the right-hand side of Figure 1 represents is that the operations associated to w'_{j+1} and v_j can be done simultaneously since no data dependencies exist and, therefore, the associated communications can be combined.

The following is an algorithmic representation of the new method.

Algorithm 3 (Delayed Classical Gram-Schmidt, DCGS2)

Input: Vector w_{j+1} to be orthogonalized against the columns of $\tilde{V}_j$
Output: Partially orthogonalized vector w'_{j+1} (with coef. h_j) and corrected vector v_j (with coef. h_{j-1})

$h_j = \tilde{V}_j^T w_{j+1}$
if $j > 1$
  $c_{j-1} = V_{j-1}^T \tilde{v}_j$
  $v_j = \tilde{v}_j - V_{j-1} c_{j-1}$
  $h_{j-1} = h_{j-1} + c_{j-1}$
end
$w'_{j+1} = w_{j+1} - V_j h_j$

Note that computed quantities such as w'_{j+1} are not equal in both algorithms since they are produced by different operations. In particular, in Figure 1 the starred arrow represents the initial orthogonalization, in which the coefficients are computed with respect to $\tilde{V}_j := [V_{j-1}, \tilde{v}_j]$, as opposed to V_j in CGS2. Therefore, in DCGS2 the coefficients of the first orthogonalization are computed against a set of vectors in which the last one is not orthogonal to full working precision. However, when the second part of the orthogonalization is carried out, the corrected vector is already available and is indeed used. Although it may seem that the referred alterations of the process would

compromise the numerical stability of the algorithm, it turns out that Algorithm 3 succeeds in computing an orthogonal set of vectors in a stable way. Comparisons with different test matrices indicate that Arnoldi with DCGS2 is numerically equivalent to Arnoldi with CGS2. However, a theoretical analysis of this equivalence remains to be done as future work.

In the CGS2 algorithm, the multiple inner product operations, $h_j = V_j^T w_{j+1}$ and $c_j = V_j^T w'_{j+1}$, can be implemented in parallel with a single reduction operation each, so two global communication operations of length $j-1$ must be performed in each Arnoldi iteration. In the case of DCGS2, the two operations $h_j = \tilde{V}_j^T w_{j+1}$ and $c_{j-1} = V_{j-1}^T \tilde{v}_j$ can be joined together, so just one reduction operation of length $2j-3$ is required. The value of j is usually very small compared to the size of the problem, so the important achievement here is the latency hiding resulting from packing all the data in a single message. This is especially important in platforms with high-latency interconnection networks. On the other hand, in this way the number of synchronization points is halved with respect to CGS2. All these improvements should lead to better parallel performance.

From the practical point of view, when implementing the DCGS2 algorithm in SLEPc, the communication merging can be easily accomplished by making use of PETSc's VecDotBegin/VecDotEnd mechanism, which gives the high-level programmer some flexibility for specifying when communications must take place.

4. Performance Analysis

In order to compare the parallel efficiency of the DCGS2 orthogonalization scheme with respect to CGS2, several test cases were analyzed in different parallel platforms. The comparison was carried out in the context of an explicitly restarted Arnoldi implementation available in SLEPc. All the tests were repeated for both orthogonalization algorithms. In all cases, the solver was requested to compute 10 eigenvalues with a maximum of 50 basis vectors. In order to minimize the effect of slight variations in the iteration count, measured wall times were divided by the number of iterations.

Two types of tests were considered. On one hand, three matrices arising from real applications (see table below) were used for measuring the parallel speed-up. This speed-up is calculated in the usual way, as the ratio of elapsed time with p processors to elapsed time with one processor. On the other hand, a synthetic test case was used for analyzing the scalability of the algorithms, measuring the scaled speed-up (with variable problem size) and Mflop/s per processor.

Name	Order	Non-zeros	Sparsity	Reference
AF23560	23,560	484,256	0.0087 %	[1]
Andrews	60,000	760,154	0.0211 %	[3]
Lin	256,000	1,011,200	0.0015 %	[3]

The first machine used for the tests was a cluster of 55 biprocessor nodes with Pentium Xeon processors at 2.8 GHz interconnected with an SCI network in a 2-D torus configuration. Only one processor per node was used in the tests reported in this section.

The speed-up measured in the Xeon cluster for the three real cases is shown in Figure 2. There are two plots for each matrix. The left one corresponds to the natural ordering of the matrix whereas the right one corresponds to a permuted version. This permutation was computed with Parmetis [6] in order to improve parallel efficiency of the matrix-vector product, which has a significant impact in the overall Arnoldi performance. The goal is to isolate as much as possible the performance of the orthogonalization part.

Both CGS2 and DCGS2 show overall good parallel performance. However, the speed-up curve corresponding to DCGS2 is always above that of CGS2, and the gap grows as the number of pro-

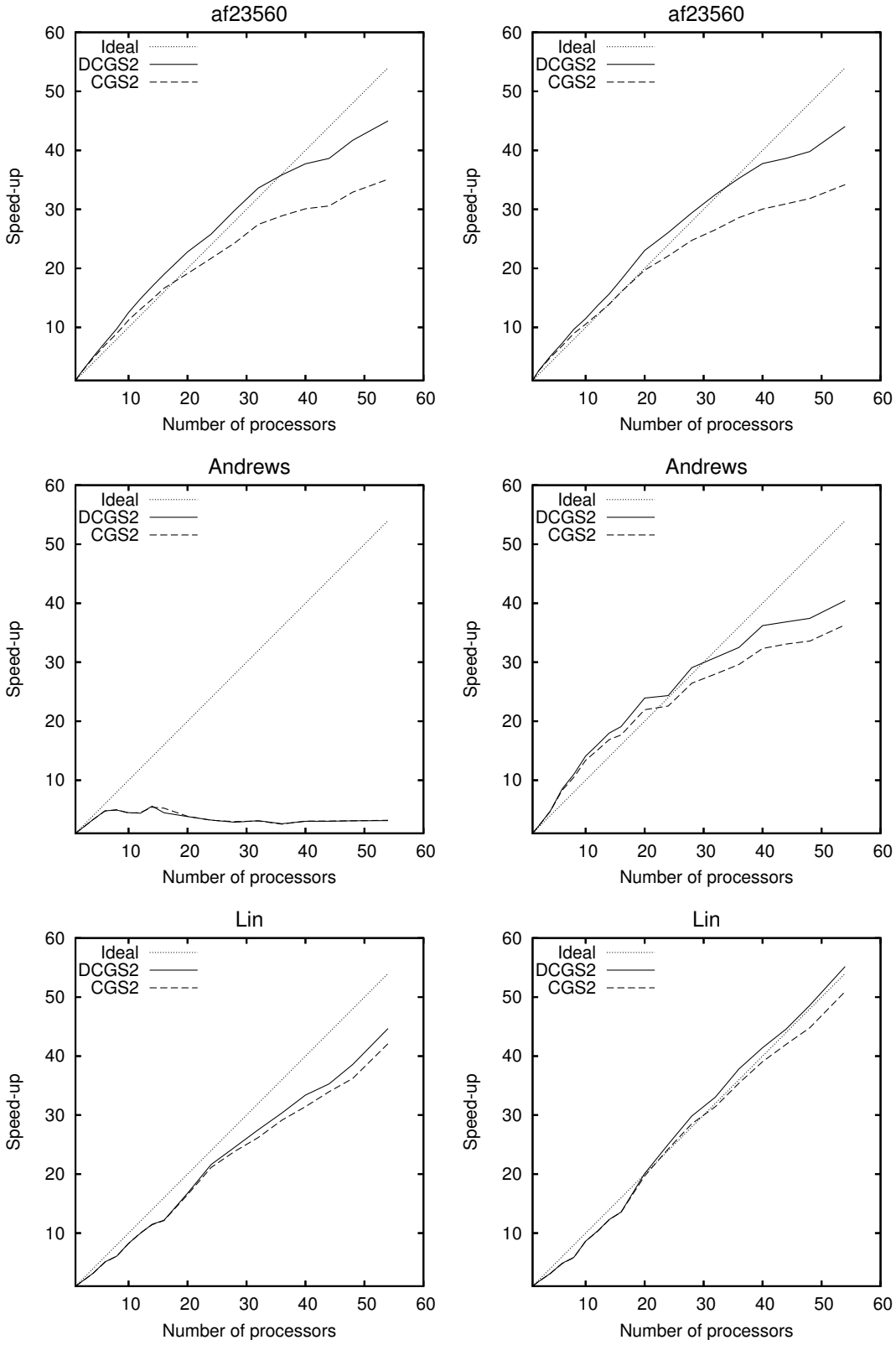


Figure 2. Measured speed-up for fixed size matrices in Xeon cluster with natural ordering (left) and with reordering (right).

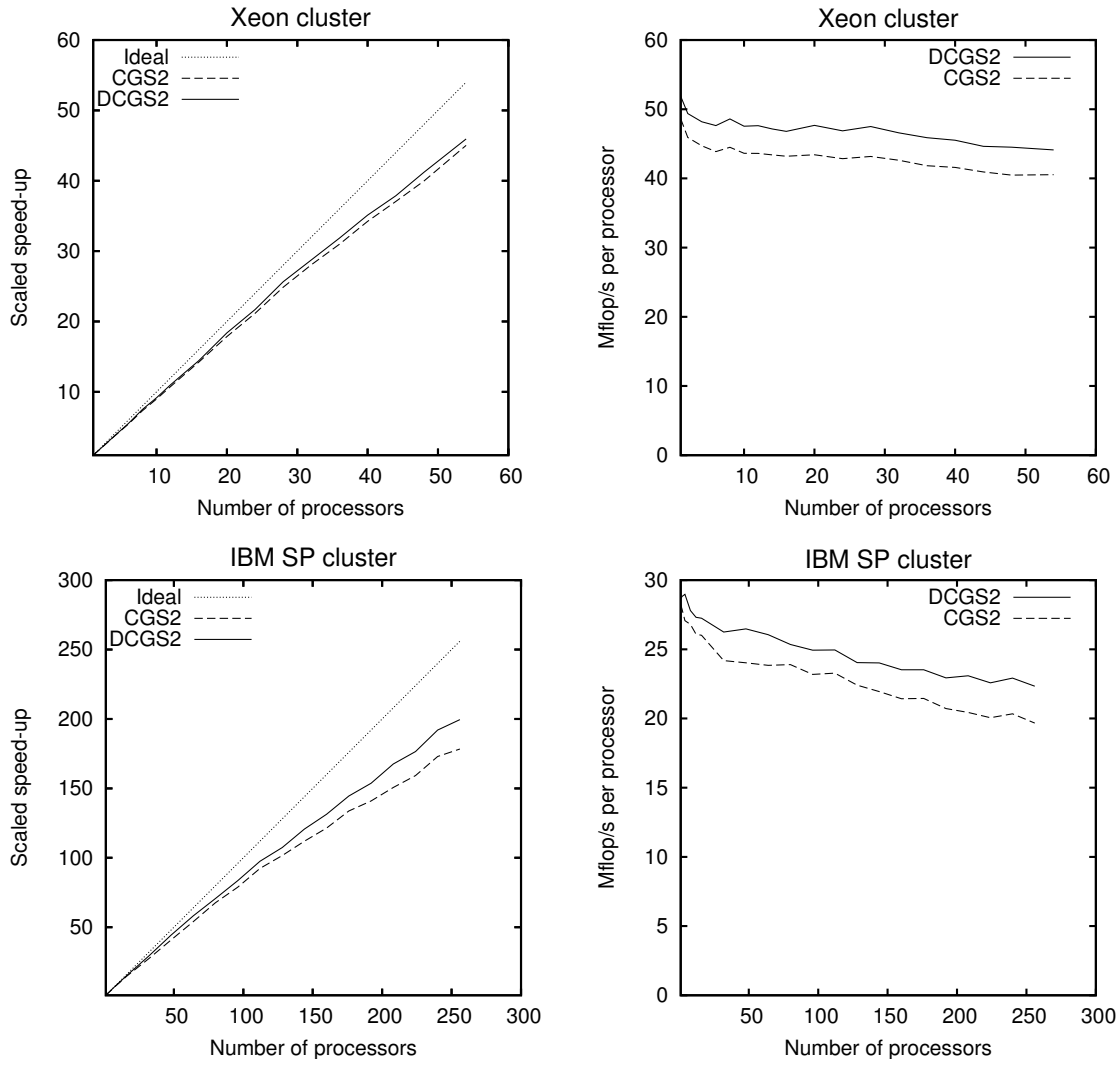


Figure 3. Scaled speed-up and Mflop/s per processor for the synthetic test case in Xeon cluster (top) and IBM SP (bottom).

cessors increases. Another observation is that there is superlinear speed-up with an intermediate number of processors. This is due to having a fixed problem size, thus decreasing the amount of data assigned to each processor and leading to better cache memory performance as the number of processors increases. This performance gain compensates for the communications cost until the local problem size fits entirely in cache memory.

To minimize this cache effect the size of local data must be kept constant, that is, the matrix dimension must grow proportionally to the number of processors, p. For this analysis, a tridiagonal matrix with random entries was used, with a dimension of $10,000 \times p$. The results in the Xeon cluster (see Figure 3) show overall good parallel performance in both alternatives, with DGCS2 having higher Mflop rate and a marginally better speed-up than CGS2.

In order to extend the analysis to more processors, this last test was repeated in an IBM SP RS/6000 computer with 380 nodes and 16 processors per node. The results up to 256 processors

are included in Figure 3 as well. As before, both variants have good parallel performance, but the DCGS2 line is significantly closer to the ideal speed-up, especially for large number of processors.

5. Discussion

In this work, a new orthogonalization algorithm called DCGS2 has been proposed in order to improve parallel efficiency in the context of Arnoldi eigensolvers. This algorithm is based on Classical Gram-Schmidt with reorthogonalization and its main goal is to reduce the number of synchronization points in parallel implementation. The performance results presented in section 4 show that the proposed algorithm achieves better efficiency in all the test cases analyzed, and scales better when increasing the number of processors.

The proposed orthogonalization technique has been tested in the context of the Arnoldi method, but it could possibly be adapted to other eigensolvers as well, such as Lanczos with full reorthogonalization or Jacobi-Davidson.

Although the DCGS2 algorithm has displayed numerical robustness in our tests, the algorithm deserves further refinement and work. In particular, a theoretical rounding error analysis should be carried out. Also, a theoretical study would be required in order to establish whether a selective reorthogonalization criterion could be applied.

Finally, as a future work, there is the possibility of further refining the parallelization of the Arnoldi method by applying the same latency hiding technique to the normalization stage in order to eliminate another synchronization point.

Acknowledgements. Part of this research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

References

[1] Z. Bai, D. Day, J. Demmel, and J. Dongarra. A test matrix collection for non-Hermitian eigenvalue problems (release 1.0). Technical Report CS-97-355, Department of Computer Science, University of Tennessee, Knoxville, TN, USA, 1997. Available at http://math.nist.gov/MatrixMarket.

[2] J. W. Daniel, W. B. Gragg, L. Kaufman, and G. W. Stewart. Reorthogonalization and stable algorithms for updating the Gram–Schmidt QR factorization. *Math. Comp.*, 30(136):772–795, October 1976.

[3] T. Davis. University of Florida Sparse Matrix Collection. NA Digest, 1992. Available at http://www.cise.ufl.edu/research/sparse/matrices.

[4] V. Hernandez, J. E. Roman, and V. Vidal. SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems. *ACM Trans. Math. Software*, 31(3), September 2005.

[5] Walter Hoffmann. Iterative algorithms for Gram-Schmidt orthogonalization. *Computing*, 41(4):335–348, 1989.

[6] G. Karypis, K. Schloegel, and V. Kumar. *ParMeTis: Parallel Graph Partitioning and Sparse Matrix Ordering Library, Version 2.0*. University of Minnesota, Dept. of Computer Science, September 1999.

[7] Y. Saad. *Numerical Methods for Large Eigenvalue Problems: Theory and Algorithms*. John Wiley, New York, 1992.

[8] Roger B. Sidje. Alternatives for parallel Krylov subspace basis computation. *Numerical Linear Algebra with Applications*, 4(4):305–331, 1997.

[9] Andreas Stathopoulos and Kesheng Wu. A block orthogonalization procedure with constant synchronization requirements. *SIAM J. Sci. Comput.*, 23(6):2165–2182, 2002.

[10] H. F. Walker. Implementation of the GMRES method using Householder transformations. *SIAM J. Sci. Statist. Comput.*, 9:152–163, 1988.

Auto-optimization of linear algebra parallel routines: the Cholesky factorization

Luis-Pedro García[a], Javier Cuenca[b], Domingo Giménez[c]

[a]Servicio de Apoyo a la Investigación Tecnológica, Universidad Politécnica de Cartagena, 30203 Cartagena, Spain

[b]Departamento de Ingeniería y Tecnología de Computadores, Universidad de Murcia, 30071 Murcia, Spain

[c]Departamento de Informática y Sistemas, Universidad de Murcia, 30071 Murcia, Spain

1. Introduction

In recent years different techniques to obtain software able to tune automatically to the conditions of parallel platforms have been studied [3,9,10,12,16]. Such developments will facilitate efficient utilization of the routines by non-expert users, e.g. those normally using linear algebra routines in the solution of large scientific or engineering problems.

One approach to obtain self-optimized linear algebra routines is the modelling of the behavior of the algorithm [5–8]. In this paper, this technique is applied to a parallel routine for the Cholesky factorization in message-passing systems. In order to obtain a good estimation with the model, the use of different costs for different types of MPI communication mechanisms (user-defined datatypes or predefined datatypes) is analyzed. The algorithm is studied in systems in which it is possible to use more than one interconnection network simultaneously. In addition, in platforms with a high ratio of the communication cost with respect to the computation cost, it is convenient to take into account that the cost of the communication parameters can vary with the volume of the communication.

The parallel Cholesky factorization is obtained with a block-cyclic partitioning in a logical two-dimensional mesh of $p = r \times c$ processes (in ScaLAPACK [4] style). An analytical model of the execution time of the parallel algorithm is developed as a function of the problem size, the system parameters (parameters of the target platform) and the algorithmic parameters. The algorithm is studied both theoretically and experimentally in order to determine the effect of the value of the system parameters on the selection of the algorithmic parameters. The typical system parameters considered in the study are the cost of arithmetic operations using BLAS kernels of levels 1, 2 or 3 (k_1, k_2, k_3) and the cost of the communication parameters (start-up, t_s, and word sending time, t_w) for the MPI library used. The algorithmic parameters are the block size, b (a block based algorithm is considered), and the parameters r and c defining the logical topology of the processes grid.

Along with the analytical model for the routine, an information system is designed and joined to the routine, providing auto-optimization capacity. The life-cycle of the auto-optimized routine can be summarized as follows [7]:

- In the design phase, the designer creates the routine if a new one is being designed. The complexity of the routine is studied, obtaining an analytical model of its execution time. Next, the installation engine and the optimization manager are created.
- In the installation phase, the system manager instals the routine, guided by the previously designed installation engine and optimization manager, and the information about each system parameter is obtained.

- In the execution phase, the optimization manager obtains information about the current system state. Next the system parameters are tuned according to the system state, and with the analytical model the optimum algorithmic parameters are selected. Finally, the routine is executed.

The remainder of the paper is structured as follows. In Section 2, the parallel Cholesky factorization is analyzed and the system parameters, the algorithmic parameters and the model for the execution time are obtained. In Section 3, experimental results are shown. Section 4 summarizes the work.

2. Parallel routine by blocks for the Cholesky factorization

In a previous work [5], a sequential block Cholesky factorization was analyzed. In that case, the block size (b) and the best library were selected to obtain execution times close to the optimum. Now a parallel implementation is considered, the $n \times n$ matrix is mapped with a block cyclic 2-D distribution onto a two-dimensional mesh of $p = r \times c$ processes.

The Cholesky factorization routine involves the following operations (figure 1 (a)):

- Process $\{0,0\}$ computes the factor L_{11} (lower triangular Cholesky factor of A_{11}).
- The processes in column 0 of the mesh compute $L_{21} = A_{21}(L_{11}^T)^{-1}$.
- All processes participate in the updating $\tilde{A}_{22} = A_{22} - L_{21}L_{21}^T = L_{22}L_{22}^T$.

In figures 1 (b) and 1 (c) the distribution of the work in the following steps is shown.

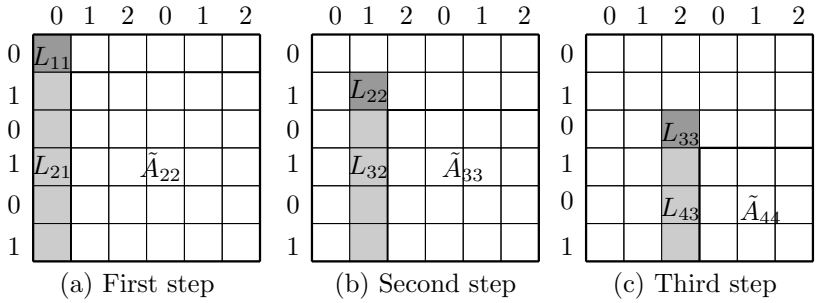


Figure 1. Work distribution in the three first steps of the parallel Cholesky routine by blocks, with $\frac{n}{b} = 6$ and $p = 2 \times 3$. The numbers on the left and on the top of the matrix represent coordinates in the 2×3 processes mesh.

The different parts of the Cholesky routine are identified in order to build the analytical model of the execution time:

- **PPOTF2** The process $\{r_i, c_j\}$, which has the $b \times b$ diagonal block A_{11}, performs the Cholesky factorization of A_{11} and computes L_{11} using the level 2 LAPACK **dpotf2** routine.
- **PTRSM** The process $\{r_i, c_j\}$ broadcasts L_{11} along the column of processes, and all the processes in the column compute the column of blocks of L_{21} by solving a triangular system of size $(n - ib) \times b$ using the level 3 BLAS **dtrsm** routine.

- **PSYRK** The column of processes with the blocks of L_{21} broadcasts columnwise their local part to the other processes in their same column, then L_{21} is broadcast rowwise. Now, all processes can update their local part of A_{22} ($\tilde{A}_{22} \leftarrow A_{22} - L_{21}L_{21}^T = L_{22}L_{22}^T$). This operation is made using BLAS routines of level 3: **dsyrk** (diagonal blocks) or **dgemm** (non diagonal blocks).

The following assumptions for the model of communications in the parallel computer are made. The parallel computer comprises a number of nodes. Each node comprises one or several identical processors interconnected by a switched communication network. The time taken to send a message of size n between any two nodes is independent of the distance between nodes and can be modelled as $t_{comm}(n,p) = t_s(p) + nt_w(n,p)$, where t_s is the latency or start-up time of the message, and t_w is the transfer time per data. The links between two nodes are full-duplex and single ported: a message can be transferred in both directions by the link at the same time, and only one message can be sent and one message can be received at the same time. Because more than a communication network can be available in the parallel computer, the values of t_s and t_w can be different between different processes, and we will have $t_{comm}(n,p)_{i,j} = t_s(p)_{i,j} + nt_w(n,p)_{i,j}$.

For this algorithm, the system parameters are: for the arithmetic cost in LAPACK and BLAS3 routines the computation cost of operations of level 2 and 3 (k_2 and k_3), and for the communication cost of the MPI library, t_s and t_w. In the sequential algorithm the only algorithmic parameter is the block size b, but in the parallel case the number of processes, p, and the dimensions of the logical two-dimensional mesh ($p = r \times c$) are additional algorithmic parameters. Thus, the values of the system parameters will depend on those of the algorithmic parameters, the problem size (n) and the library used (l). Since different level 2 and 3 routines are used, the cost with each one of them can be different and different parameters are considered: $k_{2,potf2}$, $k_{3,trsm}$, $k_{3,gemm}$ and $k_{3,syrk}$. If the algorithm uses different types of communications, the cost of the communication parameters varies. In order to show this variation, different types of broadcast communications are considered. The communication of blocks between processes on the same column is performed with MPI derived data types [14] with a cost $t_s + b^2 t_{w_d}$, and the communication of columns of blocks between processes rows is performed with MPI predefined data types, with cost $t_s + b^2 t_{w_s}$.

Thus, the execution time can be modelled by the formulas:

$$\begin{aligned} t_{arit} = {} & k_{2,dpotf2}\frac{nb^2}{3} + k_{3,dtrsm}\left[\frac{n}{r}(r-1) + \frac{n}{2}\left(\frac{n}{rb}-1\right)\right]b^2 \\ & + k_{3,dsyrk}\left\lceil\frac{1}{\sqrt{p}}\right\rceil\left(\frac{n^2-nb}{2}\right)(b+1) + \frac{2}{p}k_{3,dgemm}\left(\frac{n^3}{6} - \frac{n^2 b}{2} + \frac{nb^2}{3}\right) \end{aligned} \tag{1}$$

for the arithmetic cost, and:

$$t_{com} = \left(\frac{n}{b}-1\right)\left(t_s + b^2 t_{w_d}\right) + \frac{n}{2b}\left(\frac{n}{b}-1\right)\left(t_s + b^2 t_{w_d}\right) + \left(\frac{n}{b}-1\right)\left(bt_s + \frac{nb}{2}t_{w_s}\right) \tag{2}$$

for the communication cost.

3. Experimental Results

Experiments have been performed on two different systems in order to study the analytical model developed:

- A network of four nodes Intel Pentium 4 (P4net) with a switch FastEthernet, enabling parallel communications between them. The MPI library used is MPICH [2].
- A network of four nodes HP AlphaServer quad processors (HPC160) using Shared Memory (HPC160smp), MemoryChannel [11] (HPC160mc) or both (HPC160smp-mc) for the communication between processes. A MPI library optimized for Shared Memory and for MemoryChannel has been used [1].

The values of the arithmetic and the communication parameters are estimated with routines performing some basic operations with the same data access scheme used in the algorithm and with routines that communicate processes in the logical mesh. In both cases, the experiments were repeated several times to obtain an average value.

In an algorithm by blocks, the block size and the library to be used must be decided. Since with the optimized version of BLAS and LAPACK the lowest execution times are obtained, only these libraries are used in the experiments whose results are shown. Therefore, the arithmetic system parameters and their dependency with respect to the algorithmic parameters are shown with BLAS optimized for Pentium 4 (BLAS4) [15] and for Alpha (CXML) [13]. The values of $k_{2,potf2}$ can be considered constants with respect to r and c, but they depend on n and b. The other arithmetic system parameters, $k_{3,dgemm}$, $k_{3,dsyrk}$ and $k_{3,dtrsm}$, can be considered as a function of only the block size. In tables 1 and 2 the values used in the model are shown.

Block size	32	64	128	256
$k_{3,dgemm}$	0,001862	0,000937	0,000572	0,000467
$k_{3,dsyrk}$	0,003492	0,001484	0,001228	0,000762
$k_{3,dtrsm}$	0,011719	0,006527	0,003785	0,002325

Table 1
Values of $k_{3,dgemm}$, $k_{3,dsyrk}$ and $k_{3,dtrsm}$ (in μsec) for different block sizes, in Pentium 4 with BLAS4.

Block size	32	64	128	256
$k_{3,dgemm}$	0,000824	0,000658	0,000610	0,000580
$k_{3,dsyrk}$	0,001628	0,001164	0,000807	0,000688
$k_{3,dtrsm}$	0,001617	0,001110	0,000841	0,000706

Table 2
Values of $k_{3,dgemm}$, $k_{3,dsyrk}$ and $k_{3,dtrsm}$ (in μsec) for different block sizes, in HPC160 with CXML.

In P4net the interconnection network is very slow in comparison with the speed of the processors. It is necessary to take into account that the word sending time, t_w, varies with the message size. Table 3 shows the values obtained experimentally for the cost of the transfer time with MPI predefined data types, t_{w_s}, between processes in a same row; and table 4 for MPI derived data type, t_{w_d}, between processes in the same column. The values of the start-up time, t_s, can be considered as a function of only the number of processes, and can be approximated by $t_s = 55\ \mu sec$ for $p = 2$ and $t_s = 121\ \mu sec$ for $p = 4$.

	Message size		
p	1500	2048	> 4000
2	0,61	0,77	0,84
4	1,22	1,45	1,68

Table 3
Values of t_{w_s} (in μsec) obtained experimentally for different message sizes and number of processes. In P4net.

In HPC160mc and HPC160smp with faster interconnection networks, t_{w_s} can be considered as a function of only the number of processes, and the measured values are approximately $t_{w_s} = 0,011$ μsec for $p = 2$ and $t_{w_s} = 0,025$ μsec for $p = 4$ in HPC160smp, and $t_{w_s} = 0,072$ μsec for $p = 2$ and $t_{w_s} = 0,14$ μsec for $p = 4$ in HPC160mc. But with derived data types it is necessary to consider that the word sending time, t_{w_d}, varies with the block size (table 4). The values of the start-up time t_s can be considered as a function of only the number of processes, and can be approximated by $t_s = 4,88$ μsec for $p = 2$ and $t_s = 9,77$ μsec for $p = 4$, in HPC160smp and in HPC160mc.

	Block size											
	P4net				HPC160smp				HPC160mc			
p	32	64	128	256	32	64	128	256	32	64	128	256
2	0,97	0,84	1,00	1,10	0,019	0,024	0,020	0,019	0,095	0,091	0,089	0,090
4	1,60	1,90	1,60	1,64	0,047	0,048	0,045	0,041	0,190	0,176	0,179	0,183

Table 4
Values of t_{w_d} (in μsec) obtained experimentally for different block sizes and number of processes. In P4net, HPC160smp and HPC160mc.

With this estimation of the parameters, the model of the execution time (equations 1 and 2) is used to determine the optimum values of the algorithmic parameters for different problem sizes.

In P4net, the best selection is to execute the routine sequentially with small and medium problem sizes, but with problem sizes greater than 4096 it is recommendable to execute the routine in parallel with $p = 2 \times 1$. Table 5 shows the experimental execution time, the theoretical execution time, and the deviation ($\frac{|t_{mod} - t_{exp}|}{t_{exp}}$) between both for different problem sizes, block sizes and logical meshes. The optimum experimental and theoretical times are highlighted. The model makes a good selection of the parameters: number of processors, dimensions of the mesh and block size.

Tables 6 and 7 show the values of the algorithmic parameters provided by our method when following the model (mod.) and the experimental optimum values (opt.). The deviation of the execution time with the parameters provided by the model and the lowest experimental time obtained varying the values of the parameters is also shown. The values of the algorithmic parameters vary for different systems and problem sizes, but with the model and with the inclusion of the cost for different types of MPI communication mechanisms a satisfactory selection of the parameters is made in all the cases. A value 0 in the deviation means the values selected by the model coincide with those with which the lowest execution time was obtained. The selected block size coincides with the optimum in 27 of the 33 cases studied, and the logical topology is selected correctly in 22 of the 28 parallel experiments. In the 10 cases where the selection of the parameters does not give the optimum, the deviation on the execution time is very low. The mean of the deviations is 5.1%.

n	b	logical mesh of processes					
		1×1	1×2	2×1	2×2	1×4	4×1
4096		Experimental time (seconds)					
	32	37,498	31,501	34,673	33,321	27,858	26,068
	64	22,708	21,061	21,337	23,002	22,246	20,913
	128	14,884	17,529	16,279	20,460	21,115	18,293
	256	**13,926**	18,000	16,265	20,840	20,959	19,130
		Theoretical time (seconds)					
	32	38,293	35,668	40,117	36,233	31,497	27,936
	64	22,712	23,480	21,676	23,689	25,354	23,728
	128	14,941	20,262	16,896	21,182	24,382	19,840
	256	**14,233**	20,620	16,992	21,748	25,291	21,327
		Deviation (%)					
	32	2	13	16	9	13	7
	64	0	11	2	3	14	13
	128	0	16	4	4	15	8
	256	2	15	4	4	21	11
5120		Experimental time (seconds)					
	32	70,231	48,747	56,032	49,407	47,625	41,101
	64	43,035	33,511	33,289	35,060	37,187	34,019
	128	28,321	27,059	24,335	29,300	31,366	27,737
	256	25,316	26,342	**23,608**	29,291	29,172	27,822
		Theoretical time (seconds)					
	32	73,024	51,938	63,636	55,254	44,296	43,994
	64	41,073	34,661	33,861	35,969	35,788	36,950
	128	28,076	27,565	25,762	30,997	32,873	30,405
	256	25,306	27,217	**24,712**	31,094	33,604	31,705
		Deviation (%)					
	32	4	7	14	12	7	7
	64	5	3	2	3	4	9
	128	1	2	6	6	5	10
	256	0	3	5	6	15	14

Table 5
Comparison of the theoretical and experimental execution times, for different block sizes and logical meshes of processes, in P4net.

Thus, we can conclude the methodology proposed can be used to obtain execution times close to the optimum without user intervention.

	$p=1$			$p=2$					$p=4$				
	opt.	mod.	dev.	opt.		mod.		dev.	opt.		mod.		dev.
n	b	b	%	b	$r \times c$	b	$r \times c$	%	b	$r \times c$	b	$r \times c$	%
512	64	64	0	64	1×2	128	1×2	2.0	128	1×4	128	1×4	0
1024	128	128	0	128	1×2	64	1×2	3.4	64	4×1	64	1×4	1.2
2048	128	128	0	128	2×1	128	2×1	0	128	1×4	128	2×2	9.6
4096	256	256	0	256	2×1	128	2×1	0.1	128	4×1	128	4×1	0
5120	256	256	0	256	2×1	256	2×1	0	128	4×1	128	4×1	0
Mean			0					1.1					2.2

Table 6
Parameters selection for the Cholesky factorization, in P4net.

	HPC160smp $p=4$					HPC160mc $p=4$					HPC160smp-mc $p=8$				
	opt.		mod.		dev.	opt.		mod.		dev.	opt.		mod.		dev.
n	b	$r \times c$	b	$r \times c$	%	b	$r \times c$	b	$r \times c$	%	b	$r \times c$	b	$r \times c$	%
512	32	4×1	32	4×1	0	32	4×1	32	2×2	81	32	2×4	32	2×4	0
1024	64	4×1	64	4×1	0	64	4×1	32	2×2	62	32	2×4	32	2×4	0
2048	64	4×1	64	4×1	0	64	4×1	32	2×2	1.3	64	2×4	32	2×4	17.2
4096	128	4×1	128	4×1	0	128	2×2	128	4×1	1.6	128	2×4	128	2×4	0
5120	128	4×1	128	4×1	0	128	2×2	128	2×2	0	64	2×4	64	2×4	0
7168	128	4×1	128	4×1	0	128	2×2	128	2×2	0	64	2×4	64	2×4	0
Mean					0					24.3					2.9

Table 7
Parameters selection for the Cholesky factorization in HPC160 with Shared Memory (HPC160smp), MemoryChannel (HPC160mc) and both (HPC160smp-mc).

4. Conclusions

In this paper we have shown that in order to model linear algebra routines to obtain self-optimized routines, it is necessary to use different costs for different types of MPI communication mechanisms (user-defined datatypes or predefined datatypes) and to use different costs for the communication parameters in systems in which it is possible to use more than one interconnection network simultaneously. In these systems it is necessary to decide, in addition to the logical topology of processes, the optimal allocation of processes by node, according to the speed of the interconnection networks. The proposed method has been applied successfully to the Cholesky factorization and may also be applied to other linear algebra routines.

Acknowledgements

This work has been partially supported by Spanish MEC and FEDER under Grant TIC2003-08238-C02-02.

References

[1] Compaq MPI: Description. `http://www.hp.com/techservers/software/cmpidesc.html`.

[2] MPICH-A Portable MPI Implementation. `http://www-unix.mcs.anl.gov/mpi/mpich`.

[3] Zizhong Chen, Jack Dongarra, Piotr Luszczek, and Kenneth Roche. Self-adapting software for numerical linear algebra and LAPACK for clusters. *Parallel Computing*, 29(11-12):1723–1743, 2003.

[4] Jaeyoung Choi, Jack J. Dongarra, L. Susan Ostrouchov, Antoine P. Petitet, David W. Walker, and R. Clint Whaley. Design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. *Sci. Program.*, 5(3):173–184, 1996.

[5] Javier Cuenca, Luis-Pedro García, Domingo Giménez, José González, and Antonio M. Vidal. Empirical modelling of parallel linear algebra routines. In Roman Wyrzykowski, Jack Dongarra, Marcin Paprzycki, and Jerzy Wasniewski, editors, *PPAM*, volume 3019 of *Lecture Notes in Computer Science*, pages 169–174. Springer, 2003.

[6] Javier Cuenca, Domingo Giménez, and José González. Modelling the behaviour of linear algebra algorithms with message-passing. In *Proceedings of the Euromicro Workshop on Parallel and Distributed Processing*, pages 282–289, Mantova, Italy, 2001.

[7] Javier Cuenca, Domingo Giménez, and José González. Architecture of an automatically tuned linear algebra library. *Parallel Comput.*, 30(2):187–210, 2004.

[8] Krister Dackland and Bo Kågström. A hierarchical approach for performance analysis of scalapack-based routines using the distributed linear algebra machine. In *PARA '96: Proceedings of the Third International Workshop on Applied Parallel Computing, Industrial Computation and Optimization*, pages 186–195, London, UK, 1996. Springer-Verlag.

[9] James Demmel, Jack Dongarra, Victor Eijkhout, Erika Fuentes, Antoine Petitet, Richard Vuduc, R. Clint Whaley, and Katherine Yelick. Self adapting linear algebra algorithms and software. In *Proceedings of the IEEE: Special Issue on Program Generation, Optimization, and Adaptation*, volume 93, February 2005.

[10] Matteo Frigo and Steven G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, volume 3, pages 1381–1384, Seattle, WA, may 1998.

[11] Richard B. Gillett. Memory channel network for PCI. *IEEE Micro*, 16(1):12–18, 1996.

[12] Takahiro Katagiri, Kenji Kise, Hiroki Honda, and Toshitsugu Yuba. Effect of auto-tuning with user's knowledge for numerical software. In *CF '04: Proceedings of the 1st conference on Computing frontiers*, pages 12–25, New York, NY, USA, 2004. ACM Press.

[13] Compaq Extended Math Library. `http://h18000.www1.hp.com/math/documentation/cxml/dxml.3dxml.html`.

[14] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI–The Complete Reference: Volume 1, The MPI Core, second edition*. The MIT Press, 1998.

[15] Intel Math Kernel Library v5.2. `http://developer.intel.com/software/products/mkl/mkl52/index.htm`.

[16] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.

A Parallel Algebraic Multigrid Preconditioner Using Algebraic Multicolor Ordering for Magnetic Finite Element Analyses

T. Mifune[a], N. Obata[a], T. Iwashita[b], and M. Shimasaki[a]

[a]Department of Electrical Engineering, Kyoto University, 6158510 Kyoto, Japan

[b]Academic Center for Computing and Media Studies, Kyoto University, 6068501 Kyoto, Japan

1. Introduction

Recently, finite element (FE) analyses of magnetic fields have played a major role in the design of various electromagnetic machines. In the analyses, most computation time is consumed by the solution of large-scale sparse linear systems of equations derived from FE formulations. Algebraic multigrid (AMG) techniques [6] are known to be good preconditioners that efficiently accelerate the convergence of basic iterative methods for linear systems of equations with sparse matrices.

This paper studies parallel processing of the AMG preconditioner for FE analyses with coloring strategies. In parallelization of the AMG preconditioner, it is important to devise parallelization of the smoother. Since AMG techniques have been developed as black-box multigrid solvers, it is desirable that parallel processing does not destroy the black-box property of the AMG techniques. We propose a parallel Gauss-Seidel (GS) smoother using algebraic multicolor (AMC) ordering and compare it with the coloring strategy we set out in a previous paper [3].

The AMG preconditioner with the AMC ordered GS (AMCGS) smoother perfectly keeps the black-box property and achieves a nearly linear speed-up with respect to multigrid iterations. However, using AMC ordering might cause the deterioration of the convergence of the preconditioned solver, compared with using a sequential GS smoother. The deterioration of the convergence depends on the coloring strategy. Numerical results demonstrate that the new coloring strategy considerably improves the performance of the parallel solver, in a magnetostatic edge-element analysis for a benchmark model.

Moreover, we present new results of the application of a parallel AMG solver to magnetostatic nodal element analysis.

2. Basic Equations and FE Formulations

In this paper, the performances of the parallel AMG preconditioners are evaluated in the two different magnetostatic FE analyses: edge-element and nodal element analyses.

The two analyses deal with the same phenomenon. In static field analysis, the nodal element analysis has a large advantage with respect to the number of unknowns of the linear equations derived from FE formulations. However, edge-element applications are important in practical uses, e.g., in mode analyses of electromagnetic fields.

2.1. Edge-Element Analyses

The basic equation in the finite edge-element analysis is

$$\nabla \times (\nu \nabla \times \boldsymbol{A}) = \boldsymbol{J}_0. \tag{1}$$

Here $\boldsymbol{A}$, $\boldsymbol{J}_0$, and ν are magnetic vector potential, impressed current density, and magnetic reluctivity, respectively.

The approximate solution of the magnetic vector potential is given by

$$\hat{\boldsymbol{A}} = \sum_i u_i \boldsymbol{w}_i^{\mathrm{e}}. \tag{2}$$

Here, $\boldsymbol{w}_i^{\mathrm{e}}$ and u_i denote the edge-element trial functions [4] and the unknowns of the same number as the edges of the FE mesh, respectively.

The Galerkin formulation of (1) leads to linear systems of equations

$$K^{\mathrm{e}}\boldsymbol{u} = \boldsymbol{f}, \tag{3}$$

$$[K^{\mathrm{e}}]_{ij} = \iiint \nu(\nabla \times \boldsymbol{w}_i^{\mathrm{e}}) \cdot (\nabla \times \boldsymbol{w}_j^{\mathrm{e}})\, dV, \tag{4}$$

$$f_i = \iiint J_0 \cdot \boldsymbol{w}_i^{\mathrm{e}}\, dV + \iint \{\boldsymbol{w}_i^{\mathrm{e}} \times (\nu\nabla \times \hat{\boldsymbol{A}})\} \cdot d\boldsymbol{S}, \tag{5}$$

where $\iiint dV$ and $\iint d\boldsymbol{S}$ denote the volume and surface integrals over the analyzed domain, respectively. The second term of the right-hand-side of (5) is decided by the boundary conditions of the problem.

2.2. Nodal Element Analyses

The basic equation in finite nodal element analysis is written by

$$-\nabla \cdot (\mu\nabla\varphi) = -\nabla \cdot (\mu \boldsymbol{T_0}), \tag{6}$$

where $\mu = 1/\nu$. The magnetic scalar potential and the current vector potential are denoted by φ and $\boldsymbol{T_0}$, respectively.

The approximate solution of φ is given by

$$\hat{\varphi} = \sum_i v_i w_i^{\mathrm{n}}, \tag{7}$$

where, w_i^{n} and v_i denote the nodal element trial functions and the unknowns of the same number as the nodes of the FE mesh, respectively. The linear system of equations to be solved is written by

$$K^{\mathrm{n}}\boldsymbol{v} = \boldsymbol{g}, \tag{8}$$

$$[K^{\mathrm{n}}]_{ij} = \iiint \mu\nabla w_i^{\mathrm{n}} \cdot \nabla w_i^{\mathrm{n}}\, dV, \tag{9}$$

$$g_i = \iiint \boldsymbol{T}_0 \cdot \nabla w_i^{\mathrm{n}}\, dV - \iint \mu w_i^{\mathrm{n}} (\boldsymbol{T}_0 - \nabla\hat{\varphi}) \cdot d\boldsymbol{S}. \tag{10}$$

The second term of the right-hand-side of (10) is decided by the boundary conditions.

3. Algebraic Multigrid Methods

Multigrid (MG) methods are known to be efficient multilevel preconditioners for linear systems of equations arising in FE analyses. The convergence of iterative solvers, e.g. the conjugate gradient (CG) method, are efficiently accelerated by MG preconditioners, utilizing the hierarchy of coarse grids.

Different from geometric MG methods, AMG methods have been developed as black-box multigrid techniques [6]. Therefore, coarse grids are automatically constructed in the AMG algorithm.

In this paper, AMG methods are used as preconditioners for CG methods.

3.1. AMG Preconditioner

The AMG preconditioning is executed by computing approximate solutions of $K^{-1}\boldsymbol{r}$ by a few AMG iterations, in each CG iteration. Here, K and $\boldsymbol{r}$ denote the coefficient matrix and the residual vectors, respectively, of the considered problem.

A two-grid AMG iteration is executed as follows. The approximate solution vector of $K^{-1}\boldsymbol{r}$ is denoted by $\boldsymbol{q}$.

1. The vector $\boldsymbol{q}$ is updated by the pre-smoother. (e.g. a forward GS sweep)

2. The vector $\boldsymbol{q}$ is updated by the coarse grid correction.

3. The vector $\boldsymbol{q}$ is updated by the post-smoother. (e.g. a backward GS sweep)

In most AMG applications, GS methods are used as pre- and post- smoothers. Here, with respect to the sequential computation, the forward GS method is applied as the pre-smoother and the backward GS method as the post-smoother.

The coarse grid correction is described by

$$\boldsymbol{q} \leftarrow \boldsymbol{q} + P(P^T K P)^{-1} P^T (\boldsymbol{r} - K\boldsymbol{q}). \tag{11}$$

P is called the prolongation operator, which defines the coarse grid.

When more than one coarse grid is utilized, the inverse of $P^T K P$ in (11) is approximately computed by the reduction of a two-grid AMG iteration. On the coarsest grid, $(P^T K P)^{-1}$ is solved by a direct method.

3.2. Prolongation for Edge-Element Analyses

A special prolongation operator was developed for magnetostatic analyses using edge-elements in [5]. The shifted coefficient matrix [2] instead of K is used with respect to the preconditioning.

3.3. Prolongation for Nodal Element Analyses

The classical AMG technique [6], which was developed for linear systems of equations with symmetric M matrices, is applied for nodal element analyses. The coefficient matrix K^{n} does not strongly violate the symmetric M property.

4. Parallel AMG Preconditioned Solver

It is easy to parallelize the matrix-vector multiplications and the inner product of two vectors, because each row can be independently computed. Therefore, computations except the preconditioning are easily parallelized in the preconditioned CG solver.

In the parallelization of the AMG preconditioning;

- Multiplications by P and P^T are easily parallelized.

- Since the number of unknowns is small enough on the coarsest grid, computation cost consumed by the direct method is negligible.

- Forward and backward GS smoothers generally include sequential computations, i.e., forward and backward substitutions.

```
for (i = 0 ; i < N ; i++){
  m = 0
  (*)
  for (j = 0 ; j < i ; j++){
    if (K[i][j] != 0 && COLOR[j] == m){
      m++;
      goto (*);
    }
  }
  COLOR[i] = m;
}
```

Figure 1. Algebraic Multicolor Ordering Algorithm 1 (AMC1)

Consequently, it is important to devise parallelization of the smoothers.

Since AMG techniques have been developed as black-box multigrid solvers, which are easily used as library software [6], it is desirable that the parallel processing of the AMG preconditioning does not destroy the black-box property.

In the previous paper, we proposed a parallel AMCGS smoother [3]. Figure 1 shows the coloring algorithm in [3] (AMC1). Here, entry (i, j) of the array K and i-th entry of the array COLOR represents the entry (i, j) of the coefficient matrix and the color number of the i-th unknown, respectively. The AMC coloring strategy guarantees that, if COLOR[i] is equal to COLOR[j], K[i][j] is equal to zero. Because any two unknowns having the same color number can be independently updated in the AMCGS sweep, the AMCGS smoother is efficiently parallelized.

Only the nonzero pattern of the coefficient matrix is utilized in the AMC ordering algorithm. This keeps the black-box property of the AMG preconditioner perfect.

The convergence behavior of the preconditioned solver might change using the AMCGS smoother, compared with using the sequential GS smoother, although the convergence does not depend on the number of processors employed.

In the AMC1 algorithm, the number of colors used is decided by the nonzero pattern of the coefficient matrix; and the number of the colors is less than the maximum of the number of nonzero entries per row.

In this paper we introduce another coloring algorithm [1], which was originally developed for a parallel ICCG solver. Figure 2 shows the coloring algorithm (AMC2). Different from AMC1 ordering, the number of colors "ncolor" in the AMC2 algorithm is set as a parameter before the execution of the algorithm. A selection of the number of colors might improve the convergence of the iterative solver.

5. Numerical Results

Figure 3 shows the sample model, TEAM (Testing Electromagnetic Analysis Methods) benchmark problem 10, which is discretized using tetrahedral elements. The numbers of tetrahedral elements, edges, and nodes are 277874, 333857, and 50082, respectively.

All computations are performed on a shared memory parallel computer, a Fujitsu PRIMEPOWER HPC2500. The Fortran codes are parallelized using the OpenMP directives, using compile option

```
m = 0
for (i = 0 ; i < N ; i++){
  (*)
  for (j = 0 ; j < i ; j++){
    if (K[i][j] != 0 && COLOR[j] == m){
      m = mod(m++, ncolor);
      goto (*);
    }
  }
  COLOR[i] = m;
}
```

Figure 2. Algebraic Multicolor Ordering Algorithm 2 (AMC2)

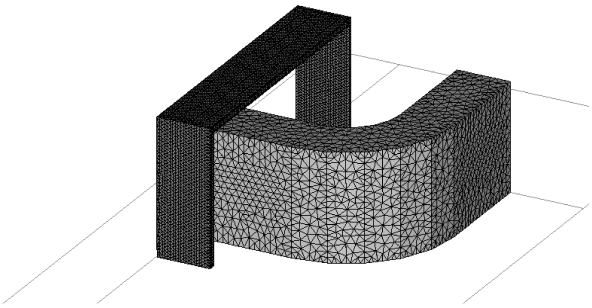


Figure 3. Example of Finite Element Mesh

"-Kfast_GP2=3 -Knolargepage -KOMP -Knoeval -X9."

In the remainder of this paper, the "sequential AMGCG" solver means the AMG preconditioned CG solver using two forward (backward) GS sweeps as the pre- (post-) smoother, which is sequentially executed. The "AMGCG-AMC1GS" mean the AMG preconditioned CG solver using two forward (backward) AMC1GS sweeps as the pre- (post-) smoother. Similarly the "AMGCG-AMC2GS" solver uses two forward (backward) AMC2GS sweeps as the pre- (post-) smoother.

5.1. Edge-Element Analysis

Table 1 presents the number of the sequential AMGCG and AMGCG-AMC1GS iterations, in the edge-element analysis. The number of colors decided by the AMC1 ordering algorithm is about 13, which is nearly constant on all grids.

Table 2 presents the number of AMGCG-AMC2GS iterations when the number of colors is set to various values. The best CG convergence is obtained when the number of colors is 70, although the number of AMGCG-AMC2GS iterations does not greatly change by the selection of the number of colors. The CG convergence, which deteriorates using AMC1 ordering, is significantly improved using the AMC2 algorithm.

Table 3 compares the elapsed time of the AMGCG solvers, in which T_{s} and T_{i} denote the times

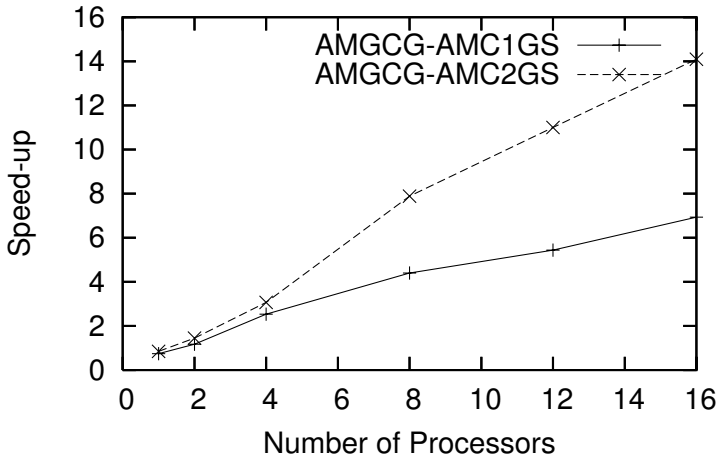


Figure 4. Speed-up of AMGCG Solvers with AMC1 and AMC2 Orderings (Edge-Element Analysis)

Table 1
Number of Sequential AMGCG and AMGCG-AMC1GS Iterations (Edge-Element Analysis)

Sequential AMGCG	AMGCG-AMC1GS
191	289

consumed by the grid construction and by preconditioned CG iterations, respectively. The number of colors of the AMC2 algorithm is set to be 70. With respect to $T_{\rm i}$, parallel AMGCG solvers attain nearly linear speed-up, due to the property that the number of processors used does not affect the convergence.

Figure 4 demonstrates performances of the parallel solvers, in which the "speed-up" is calculated by $(T_{\rm s} + T_{\rm i})_{|\rm SequentialAMGCG}/(T_{\rm s} + T_{\rm i})$. A very good performance is achieved using the AMC2 ordering algorithm.

5.2. Nodal Element Analysis

Table 4 gives the number of sequential AMGCG and AMGCG-AMC1GS iterations, in the nodal element analysis. The maximum number of colors is 31 in the AMC1 algorithm.

Table 5 presents the number of the AMGCG-AMC2GS iterations with respect to various numbers of colors, "ncolor" in Figure 2. The number of AMGCG-AMC2GS iterations does not change in the table.

Table 6 compares the elapsed time. The number of colors of the AMC2 algorithm is set to 60. Parallel solvers achieve good speed-up with respect to $T_{\rm i}$, although the sequential parts $(T_{\rm s})$ occupy most of the total time when using 4 processors or more.

Table 2
Number of AMGCG-AMC2GS Iterations (Edge-Element Analysis)

Number of colors	Number of AMGCG-AMC2GS Iterations
60	195
70	193
80	195
90	195

Table 3
Elapsed Time of the AMGCG Solvers (Edge-Element Analysis)

Number of Processors	AMGCG-AMC1GS		AMGCG-AMC2GS		Sequential AMGCG	
	T_{s} [s]	T_{i} [s]	T_{s} [s]	T_{i} [s]	T_{s} [s]	T_{i} [s]
1	5.49	928.22	6.40	816.22	5.40	685.7
2	5.18	585.38	7.23	471.13	-	-
4	4.67	268.90	6.03	218.99	-	-
8	4.22	153.03	5.77	81.91	-	-
16	4.18	95.54	5.54	43.34	-	-

Table 4
Number of Sequential AMGCG Iterations and AMGCG-AMC1GS Iterations (Nodal Element Analysis)

Sequential AMGCG	AMGCG-AMC1GS
17	17

Table 5
Number of AMGCG-AMC2GS Iterations (Nodal Element Analysis)

Number of colors	Number of AMGCG-AMC2GS Iterations
60	17
70	17
80	17
90	17

Table 6
Elapsed Time of the AMGCG Solvers (Nodal Element Analysis)

Number of Processors	AMGCG-AMC1GS		AMGCG-AMC2GS		Sequential AMGCG	
	T_{s} [s]	T_{i} [s]	T_{s} [s]	T_{i} [s]	T_{s} [s]	T_{i} [s]
1	2.79	4.51	3.78	4.66	2.92	4.03
2	3.03	2.83	3.02	2.06	-	-
4	2.88	1.38	3.19	1.27	-	-
8	2.98	0.88	3.29	0.77	-	-
16	2.56	0.58	3.18	0.46	-	-

6. Conclusion

For efficient parallelization of the AMG preconditioner, we introduce an AMC2 algorithm different from that used in previous work. In edge-element magnetostatic analysis, it is demonstrated that the CG convergence is significantly improved using the AMC2 algorithm.

Numerical results using the nodal elements are also presented. The parallel AMG efficiency is not good because the AMG setup process, which is sequentially executed, consumes a large part of the total solution time. However, the speed-up of the iteration part is very good. The parallelization by AMC ordering will be effective in time-dependent nodal element analyses, in which the setup time is negligible.

7. Acknowledgement

Computation in the present paper was carried out as a collaborative research project using the KDK system of the Research Institute for Sustainable Humanosphere (RISH) at Kyoto University.

References

[1] T. Iwashita and M. Shimasaki. Algebraic multicolor ordering for parallelized ICCG solver in finite-element analyses. *IEEE Trans. Magn.*, 38(2):429–432, 2002.

[2] T. Mifune, T. Iwashita, and M. Shimasaki. New algebraic multigrid preconditioning for iterative solvers in electromagnetic finite edge-element analyses. *IEEE Trans. Magn.*, 39(3):1677–1680, 2003.

[3] T. Mifune, T. Iwashita, and M. Shimasaki. A parallel algebraic multigrid solver for fast magnetic edge-element analyses. *IEEE Trans. Magn.*, 41(5):1660–1663, 2005.

[4] J. Nédélec. A new family of mixed finite elements in R^3. *Numer. Math.*, 38:57–81, 1986.

[5] S. Reitzinger and J. Schoberl. An algebraic multigrid method for finite element discretizations with edge elements. *Numer. Linear Algebra Appl.*, 9:223–238, 2002.

[6] J. Ruge and K. Stüben. Algebraic multigrid. In S. McCormick, editor, *Multigrid Methods*, volume 3 of *Frontiers in Applied Mathematics*, pages 73–130. SIAM, Philadelphia, 1987.

Parallel Newton-type iterative methods based on ILU factorizations

Josep Arnal[a], Héctor Migallón[b], Violeta Migallón[a], José Penadés[a]

[a]Departamento de Ciencia de la Computación e Inteligencia Artificial,
Universidad de Alicante, E-03071 Alicante, Spain, {arnal, violeta, jpenades}@dccia.ua.es

[b]Departamento de Física y Arquitectura de Computadores,
Universidad Miguel Hernández, 03202 Elche, Alicante, Spain, hmigallon@umh.es

Parallel iterative algorithms based on the Newton method and on two of its variations, the Shamanskii method and the Chord method, for solving nonlinear systems are proposed. These algorithms are based on the two–stage multisplitting methods. Concretely, in order to construct the inner splittings, incomplete LU factorizations are considered. Convergence properties of these parallel methods are analyzed and computational results on two parallel computing systems are discussed. In order to illustrate the behaviour of the proposed algorithms, we have considered a nonlinear elliptic partial differential equation, known as the Bratu problem, which comes from a simplification of the solid fuel ignition model in thermal combustion theory. The reported experiments show the effectiveness of these methods.

1. Introduction

Consider the problem of solving a nonlinear system of the form $F(x) = 0$, where $F : \mathbb{R}^n \to \mathbb{R}^n$ is a nonlinear mapping. Considering that there exists a solution x^* of this system, we can use for solving it the classical Newton method (cf. [10], [7]). Given an initial vector $x^{(0)}$, this method produces the following sequence of vectors

$$x^{(\ell+1)} = x^{(\ell)} - \delta_x^{(\ell)}, \quad \ell = 0, 1, \ldots, \tag{1}$$

where $\delta_x^{(\ell)}$ is the solution of the linear system

$$F'(x^{(\ell)})z = F(x^{(\ell)}), \tag{2}$$

and $F'(x)$ denotes the Jacobian matrix.

Iterative methods can be used for the solution of (2). In this case we are in the presence of a Newton iterative method. Descriptions of these methods can be found, e.g., in [10]. In order to generate efficient algorithms to solve the nonlinear system $F(x) = 0$ on a parallel computer, in [1] it was constructed a parallel Newton iterative algorithm, in which the approximations of the linear systems (2) are accomplished by using a two–stage multisplitting method [5].

In order to construct these methods, let us consider for each x, a two–stage multisplitting of $F'(x)$, $\{P_k(x), Q_k(x), M_k(x), N_k(x), E_k\}_{k=1}^p$, that is, a collection of matrices such that $F'(x) = P_k(x) - Q_k(x),\ 1 \le k \le p$, are splittings of $F'(x)$, called outer splittings, $P_k(x) = M_k(x) - N_k(x),\ 1 \le k \le p$, are splittings of $P_k(x)$, called inner splittings, and $E_k,\ 1 \le k \le p$, are diagonal nonnegative weighting matrices such that $\sum_{k=1}^p E_k = I$.

Let us further consider two sequences of integers. The sequence $m_\ell,\ \ell = 0, 1, \ldots$, indicates the number of linear steps performed to approximate the linear system (2) at the global nonlinear iteration ℓ, and the sequence of non–stationary parameters $q(\ell, s, k),\ \ell = 0, 1, \ldots,\ s = 1, 2, \ldots, m_\ell,\ 1 \le k \le p$, indicates the number of inner steps which processor k carries out at the s outer linear step and at the global iteration ℓ.

Given an initial vector $x^{(0)}$, that satisfies some initial conditions that we will analyze later on, we construct a sequence of vectors $\{x^{(\ell)}\}_{\ell=0}^{\infty}$ in the following way. In order to approximate the linear system $F'(x^{(\ell)})z = F(x^{(\ell)}),\ \ell = 0, 1, \ldots$, setting $z^{(0)} = 0$, we compute the iterates $z^{(s)} = \sum_{k=1}^{p} E_k z_k^{(q(\ell,s,k))},\ s = 1, 2, \ldots, m_\ell$, where $z_k^{(q(\ell,s,k))}$ is an approximation to the solution of the linear system $P_k(x^{(\ell)})z_k = Q_k(x^{(\ell)})z^{(s-1)} + F(x^{(\ell)})$, obtained by computing $q(\ell, s, k)$ iterations of the iterative method based on the splitting $P_k(x^{(\ell)}) = M_k(x^{(\ell)}) - N_k(x^{(\ell)})$, taking as initial vector $z_k^{(0)} = z^{(s-1)}$. That is, $z_k^{(q(\ell,s,k))}$ is achieved by performing the following $q(\ell, s, k)$ iterations, $z_k^{(m+1)} = M_k(x^{(\ell)})^{-1}N_k(x^{(\ell)})z_k^{(m)} + M_k(x^{(\ell)})^{-1}\left(Q_k(x^{(\ell)})z^{(s-1)} + F(x^{(\ell)})\right),\ m = 0, 1, \ldots, q(\ell, s, k) - 1$.

Finally, the next global iterate is computed as

$$x^{(\ell+1)} = x^{(\ell)} - \delta_x^{(\ell)},\ \ell = 0, 1, \ldots, \text{ where } \delta_x^{(\ell)} = z^{(m_\ell)}. \tag{3}$$

In order to study the convergence of this method, let us denote, for $\ell = 0, 1, \ldots,\ s = 1, 2, \ldots, m_\ell$, $1 \le k \le p$,

$$T_k^{\ell,s}(x) = \left(M_k(x)^{-1}N_k(x)\right)^{q(\ell,s,k)} + \sum_{i=0}^{q(\ell,s,k)-1} \left(M_k(x)^{-1}N_k(x)\right)^i M_k(x)^{-1}Q_k(x) \tag{4}$$

$$H_{\ell,s}(x) = \sum_{k=1}^{p} E_k T_k^{\ell,s}(x),\ \ B_{\ell,s}(x) = \sum_{k=1}^{p} E_k \sum_{i=0}^{q(\ell,s,k)-1} \left(M_k(x)^{-1}N_k(x)\right)^i M_k(x)^{-1} \tag{5}$$

Thus $z^{(s)} = H_{\ell,s}(x^{(\ell)})z^{(s-1)} + B_{\ell,s}(x^{(\ell)})F(x^{(\ell)}),\ s = 1, 2, \ldots, m_\ell$, with $z^{(0)} = 0$. From this expression it is easy to deduce $z^{(s)} = \left(\sum_{i=1}^{s-1} \prod_{j=i+1}^{s} H_{\ell,j}(x^{(\ell)})B_{\ell,i}(x^{(\ell)}) + B_{\ell,s}(x^{(\ell)})\right) F(x^{(\ell)})$, where $\prod_{j=i+1}^{s} H_{\ell,j}(x)$ denotes the product of the matrices $H_{\ell,j}(x)$, in the order $H_{\ell,s}(x) \cdots H_{\ell,i+1}(x)$. With this notation the parallel Newton two–stage iterative method (3) can be written as follows

$$x^{(\ell+1)} = G_{\ell,m_\ell}(x^{(\ell)}),\ \ell = 0, 1, \ldots, \tag{6}$$

where $G_{\ell,m_\ell}(x) = x - A_{\ell,m_\ell}(x)F(x)$, and

$$A_{\ell,m_\ell}(x) = \sum_{i=1}^{m_\ell-1} \prod_{j=i+1}^{m_\ell} H_{\ell,j}(x)B_{\ell,i}(x) + B_{\ell,m_\ell}(x). \tag{7}$$

The experiments displayed in [1] show the good behaviour of these methods. In Section 2 we analyze the convergence of these methods when incomplete LU (ILU) factorizations are used in order to obtain the inner splittings. Moreover, in order to reduce the computational cost of each non–linear iteration, we describe and analyze two acceleration techniques for this parallel Newton iterative method, based on both the Chord method and the Shamanskii method. In Section 3 we present some numerical experiments, which illustrate the performance of these algorithms. Previously, in the rest of this section we present some notation, definitions and preliminary results to which we refer later.

A matrix A is said to be a nonsingular M–matrix if A has all nonpositive off–diagonal entries and it is monotone, i.e., $A^{-1} \ge O$. For any matrix $A = (a_{ij}) \in \mathbb{R}^{n\times n}$, we define its comparison matrix $\langle A\rangle = (\alpha_{ij})$ by $\alpha_{ii} = |a_{ii}|,\ \alpha_{ij} = -|a_{ij}|,\ i \ne j$. The matrix A is said to be an H–matrix if $\langle A\rangle$ is a nonsingular M–matrix. The splitting $A = M - N$ is called a regular splitting if $M^{-1} \ge O$ and $N \ge O$, it is called a weak regular splitting if $M^{-1} \ge O$ and $M^{-1}N \ge O$; the splitting is an H–splitting if $\langle M\rangle - |N|$ is an M–matrix, the splitting is an H–compatible splitting if $\langle A\rangle = \langle M\rangle - |N|$; see e.g., [4]. Let x be a positive vector, we consider the vector norm $\|y\|_x = \inf\{\beta > 0 \ :\ |y| \le \beta x\}$. This vector norm is monotone increasing and for every matrix $B \in \mathbb{R}^{n\times n}$ it satisfies $\|\ |B|x\|_x = \|B\|_x$, where $\|B\|_x$ denotes the matrix norm of B induced by the vector norm above defined.

Lemma 1 [6] *Let $A = M - N$ be an H–splitting. Then A and M are H–matrices and $\rho(M^{-1}N) \leq \rho(\langle M\rangle^{-1}|N|) < 1$.*

Lemma 2 [6] *Let $A = M - N$ be an H–compatible splitting and assume that A is an H–matrix. Then $A = M - N$ is an H–splitting and thus the splitting is convergent.*

Theorem 1 [11] *Let A be an $n \times n$ M–matrix, then for every zero pattern subset S of $S_n = \{(i,j) : i \neq j,\ 1 \leq i,j \leq n\}$, there exist a unit lower triangular matrix $L = (l_{ij})$, an upper triangular matrix $U = (u_{ij})$, and a matrix $N = (n_{ij})$ with $l_{ij} = u_{ij} = 0$ if $(i,j) \in S$ and $n_{ij} = 0$ if $(i,j) \notin S$, such that $A = LU - N$ is a regular splitting of A. Moreover, the factors L and U are unique.*

Theorem 2 [9] *Let A be an $n \times n$ H–matrix. Let $A = LU - N$ and $\langle A\rangle = \hat{L}\hat{U} - \hat{N}$ be the ILU factorizations of A and $\langle A\rangle$ corresponding to a zero pattern subset S of $S_n = \{(i,j) :\ i \neq j,\ 1 \leq i,j \leq n\}$, respectively. Then $|L^{-1}| \leq \hat{L}^{-1},\ |U^{-1}| \leq \hat{U}^{-1},\ |(LU)^{-1}N| \leq (\hat{L}\hat{U})^{-1}\hat{N}$.*

2. Convergence

In order to obtain the two–stage multisplitting of the Jacobian matrix $F'(x)$, we consider the following outer splittings $F'(x) = P_k(x) - Q_k(x),\ 1 \leq k \leq p$. Then we perform ILU factorizations of the matrices $P_k(x)$. This entails for each $k,\ 1 \leq k \leq p$, a decomposition of the form $P_k(x) = M_k(x) - N_k(x)$, where $M_k(x) = L_k(x)U_k(x)$, and the matrices $L_k(x)$ and $U_k(x)$ are unit lower triangular and upper triangular matrices, respectively, and $N_k(x)$ is the residual or error of the factorization. This incomplete factorization is rather easy to compute. A general algorithm for building ILU factorizations can be derived by performing Gaussian elimination and dropping some elements in predetermined nondiagonal positions (see e.g., [11]). As was done in [1], to study the convergence of the iterative scheme (6) setting $M_k(x) = L_k(x)U_k(x)$, we need to make the following assumptions:

(i) There exists an $r_0 > 0$ such that F is differentiable on $\mathcal{S}_0 \equiv \{x \in \mathbb{R}^n\ :\ \|x - x^*\| < r_0\}$,

(ii) the Jacobian matrix at x^*, $F'(x^*)$, is nonsingular,

(iii) there exists a $\vartheta > 0$ such that for $x \in \mathcal{S}_0,\ \|F'(x) - F'(x^*)\| \leq \vartheta\|x - x^*\|$,

(iv) $P_k(x)$ and $M_k(x),\ 1 \leq k \leq p$, are Lipschitz–continuous at x^*, i.e., there exist positive constants $\mu_k,\ \eta_k,\ 1 \leq k \leq p$, such that, for $x \in \mathcal{S}_0$, $\|P_k(x) - P_k(x^*)\| \leq \mu_k\|x - x^*\|$, $\|M_k(x) - M_k(x^*)\| \leq \eta_k\|x - x^*\|$,

(v) $P_k(x^*)$ and $M_k(x^*),\ 1 \leq k \leq p$, are nonsingular,

(vi) there exists $0 \leq \alpha < 1$, such that, for each positive integer s and $\ell = 0, 1, \ldots$, $\|H_{\ell,s}(x^*)\| \leq \alpha$, where $H_{\ell,s}(x)$ is defined in (5).

Theorem 3 [1] *Let assumptions (i)–(vi) hold and $F(x^*) = 0$. Let $\{m_\ell\}_{\ell=0}^{\infty}$ be a sequence of positive integers, and define*

$$m = \max\left[\{m_0\} \cup \left\{m_\ell - \sum_{i=0}^{\ell-1} m_i\ :\ \ell = 1, 2, \ldots\right\}\right]. \tag{8}$$

Suppose that $m < +\infty$ and that the sequence of number of inner linear iterations $q(\ell, s, k),\ \ell = 0, 1, \ldots,\ s = 1, 2, \ldots, m_\ell,\ 1 \leq k \leq p$, is bounded by $q > 0$. Then, there exist $r > 0$ and $c < 1$ such that, for $x^{(0)} \in \mathcal{S} \equiv \{x \in \mathbb{R}^n\ :\ \|x - x^*\| < r\}$, the sequence of iterates defined by (6) converges to x^* and satisfies $\|x^{(\ell+1)} - x^*\| \leq c^{m_\ell}\|x^{(\ell)} - x^*\|$.

We note that if the inner splittings satisfy $\|M_k(x^*)^{-1}N_k(x^*)\| < 1,\ 1 \leq k \leq p$, Theorem 3 remains valid without the assumption that the sequence of number of inner linear iterations $q(\ell, s, k)$, $\ell = 0, 1, \ldots,\ s = 1, 2, \ldots, m_\ell,\ 1 \leq k \leq p$, be bounded (see Remark 4.3 of [1]).

Theorem 4 *Let assumptions (i)–(iv) hold and $F(x^*) = 0$. Let $\{m_\ell\}_{\ell=0}^{\infty}$, be a sequence of positive integers, and define m as in (8). Suppose that $m < +\infty$. Let $F'(x^*)$ be an H–matrix, and the splittings $F'(x^*) = P_k(x^*) - Q_k(x^*),\ 1 \leq k \leq p$, be H–compatible. Let $P_k(x^*) = M_k(x^*) - N_k(x^*),\ 1 \leq k \leq p$, where $M_k(x^*) = L_k(x^*)U_k(x^*)$ is the ILU factorization of $P_k(x^*)$ corresponding to a zero pattern subset S_k of $S_n = \{(i, j) :\ i \neq j,\ 1 \leq i, j \leq n\}$. Then, there exist $r > 0$ and $c < 1$ such that, for $x^{(0)} \in \mathcal{S} \equiv \{x \in \mathbb{R}^n\ :\ \|x - x^*\| < r\}$, the sequence of iterates defined by (6) converges to x^* and satisfies $\|x^{(\ell+1)} - x^*\| \leq c^{m_\ell}\|x^{(\ell)} - x^*\|$.*

Proof: From Theorem 3 it follows that it suffices to prove that there exists $\alpha < 1$ such that $\|(L_k(x^*)U_k(x^*))^{-1}N_k(x^*)\| \leq \alpha,\ 1 \leq k \leq p$, and $\|H_{\ell,s}(x^*)\| \leq \alpha,\ \ell = 0, 1, 2, \ldots,\ s = 1, 2, \ldots, m_\ell$, for some matrix norm.

Since $F'(x^*) = P_k(x^*) - Q_k(x^*),\ 1 \leq k \leq p$, are H–compatible splittings of an H–matrix, from Lemmata 1 and 2, it follows that $P_k(x^*),\ 1 \leq k \leq p$, are H–matrices. Then $\langle P_k(x^*)\rangle,\ 1 \leq k \leq p$, are M–matrices. Following Theorem 1 there exists an ILU factorization of $\langle P_k(x^*)\rangle$ corresponding to the zero pattern subset S_k such that $\langle P_k(x^*)\rangle = \hat{L}_k(x^*)\hat{U}_k(x^*) - \hat{N}_k(x^*),\ 1 \leq k \leq p$, are regular splittings.

Let us denote $\hat{R}_k(x) = (\hat{L}_k(x)\hat{U}_k(x))^{-1}\hat{N}_k(x)$, then taking into account (4) y (5), from Theorem 2 it follows

$$|H_{\ell,s}(x^*)| \leq \sum_{k=1}^{p} E_k \hat{R}_k^{q(\ell,s,k)}(x^*) + \sum_{k=1}^{p} E_k \left(\sum_{i=0}^{q(\ell,s,k)-1} \hat{R}_k^i(x^*) \right) (\hat{L}_k(x^*)\hat{U}_k(x^*))^{-1}|Q_k(x^*)| \equiv \hat{H}_{\ell,s}(x^*).$$

That is, $|H_{\ell,s}(x^*)| \leq \hat{H}_{\ell,s}(x^*)$, where $\hat{H}_{\ell,s}(x^*)$ are the iteration matrices of a two–stage multisplitting method for the splittings $\langle F'(x^*)\rangle = \langle P_k(x^*)\rangle - |Q_k(x^*)|,\ 1 \leq k \leq p$ and $\langle P_k(x^*)\rangle = \hat{L}_k(x^*)\hat{U}_k(x^*) - \hat{N}_k(x^*),\ 1 \leq k \leq p$. Moreover these splittings are regular.

We consider the vector $e = (1, \ldots, 1)^T$, since $\langle F'(x^*)\rangle^{-1} \geq O$, it follows that $u = \langle F'(x^*)\rangle^{-1}e > 0$, and we can assure (see e.g., [5]) that there exists $0 \leq \alpha < 1$ such that $|H_{\ell,s}(x^*)|u \leq \hat{H}_{\ell,s}(x^*)u \leq \alpha u$. Hence $\|H_{\ell,s}(x^*)\|_u \leq \alpha,\ \ell = 0, 1, \ldots,\ s = 1, \ldots, m_\ell$. On the other hand, setting the fixed vector $u = \langle F'(x^*)\rangle^{-1}e$, it obtains $(\hat{L}_k(x^*)\hat{U}_k(x^*))^{-1}\hat{N}_k(x^*)u \leq u - (\hat{L}_k(x^*)\hat{U}_k(x^*))^{-1}e < u$.

Then, from Theorem 2 it follows that $|(L_k(x^*)U_k(x^*))^{-1}N_k(x^*)|u < u$, and therefore for $1 \leq k \leq p$, $\|(L_k(x^*)U_k(x^*))^{-1}N_k(x^*)\|_u < 1$. Thus the proof is complete.

In the parallel Newton two–stage method (6), when ILU factorizations are used in order to construct the two–stage multisplitting, the Jacobian matrix must be computed at the current iterate, and a portion of this matrix ($P_k(x^{(\ell)})$) must be factored at each non–linear iteration. One approach to reduce the cost of each non–linear iteration is to consider the sequence of iterates $x^{(\ell+1)} = \hat{G}_{\ell,m_\ell}(x^{(\ell)}, x^{(0)})$, where $\hat{G}_{\ell,m_\ell}(x, y) = x - A_{\ell,m_\ell}(y)F(x)$ and $A_{\ell,m_\ell}(y)$ is defined in (7).

This method, based on the Chord method (see e.g., [10]), will be called the parallel Chord two–stage method. The only difference in implementation from the parallel Newton two–stage method is that the computation and, therefore, the obtaining of the two–stage multisplitting of the Jacobian matrix is done before the nonlinear iteration begins. This technique could be interesting to reduce the computational time. The difference in the iteration itself is that another approximation to $F'(x^{(\ell)})$ is used. Another technique consists of alternating a Newton step with a sequence of Chord steps. In this case we can describe the transition from $x^{(\ell)}$ to $x^{(\ell+1)}$ by

For $\ell = 0, 1, 2, \ldots$
If mod$(\ell, \tau) = 0$ then $y = x^{(\ell)}$
$x^{(\ell+1)} = \hat{G}_{\ell,m_\ell}(x^{(\ell)}, y)$.

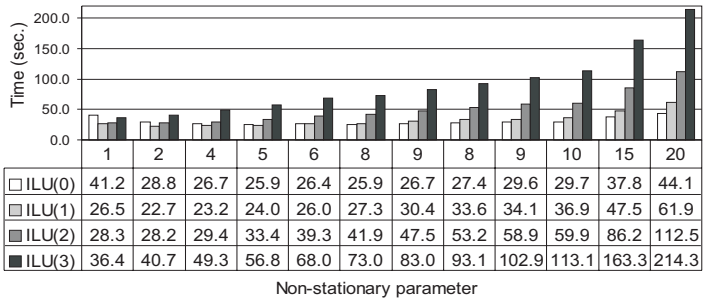
This method, based on the Shamanskii method (see e.g., [10]), will be called the parallel Shamanskii two–stage method. Note that $\tau = 1$ is the parallel Newton two–stage method and $\tau = \infty$ is the corresponding Chord method.

It is not difficult to see that the convergence results of Theorem 3, and therefore, those of Theorem 4, remain valid for both variations of the Newton method described above.

3. Numerical experiments

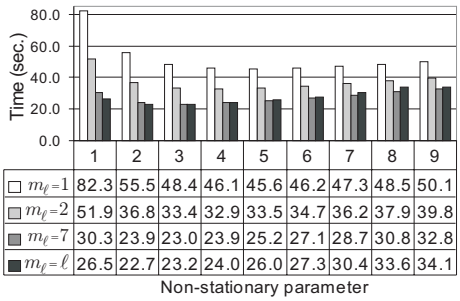
In order to illustrate the behaviour of these methods, we implemented the algorithms described here on two distributed memory multiprocessors. The first platform is an IBM RS/6000 SP with 8 nodes. These nodes are 120 MHz Power2 Super Chip and are connected through a high performance switch with latency time of 40 microseconds and a bandwidth of 110 Mbytes per second. The second platform is an Ethernet network of 6 Pentiums IV connected through a switch with a bandwidth of 1 Gbit per second (Cluster 1 Gb/sec.). The parallel environment has been managed using the MPI library of parallel routines. Moreover, we have used the BLAS routines for vector computations and the SPARSKIT routines for handling sparse matrices. In order to illustrate the behaviour of the above algorithms, we have considered a nonlinear elliptic partial differential equation, known as the Bratu problem. In this problem, heat generation from a combustion process is balanced by heat transfer due to conduction. The model problem is given as $\nabla^2 u - \lambda e^u = 0$, where u is the temperature and λ is a constant known as the Frank–Kamenetskii parameter (see e.g., [3]). There are two possible steady–state solutions to this problem for a given value of λ. One solution is close to $u = 0$ and it is easy to obtain. A close starting point is needed to converge to the other solution. For our model case, we consider a 3D square domain Ω of unit length and $\lambda = 6$. To solve our model problem using the finite difference method, we consider a grid in Ω of d^3 nodes. This discretization yields a nonlinear system of the form $F(x) = Ax + \Phi(x) - b = 0$, where $\Phi : \mathbb{R}^n \to \mathbb{R}^n$ is a nonlinear diagonal mapping (i.e., the ith component Φ_i of Φ is a function only of x_i). The Jacobian matrix is a sparse matrix of order d^3 and the typical number of nonzero elements per row of this matrix is seven, with fewer in rows corresponding to boundary points of the physical domain. In our experiments, we have considered the outer splittings $F'(x) = P_k(x) - Q_k(x)$ determined by $P_k(x) = \text{diag}(I, \ldots, D_k(x), \ldots, I),\ 1 \le k \le p$, where $D_k(x)$ consists of the k diagonal block of $F'(x)$ of size n_k, and the inner splittings $P_k(x) = M_k(x) - N_k(x)$ are determined by $M_k(x) = \text{diag}(I, \ldots, L_k(x)U_k(x), \ldots, I),\ 1 \le k \le p$, where $L_k(x)U_k(x)$ is the "level of fill–in" factorizations ILU of $D_k(x)$ [11]. Each diagonal weighting matrix E_k has ones in the entries corresponding to the diagonal block $D_k(x)$ and zero otherwise. Note that, with this choice of the two-stage multisplitting, each processor k only needs to approximate, at each inner iteration, linear systems of size n_k, where $\sum_{k=1}^{p} n_k = n$, and n is the size of the problem to be solved. In order to preserve the block structure of the Jacobian matrices we have considered, in our experiments, n_k a multiple of d.

Let us denote by ILU(S) the incomplete LU factorization associated with the zero pattern subset S of $S_n = \{(i, j) :\ i \neq j,\ 1 \le i, j \le n\}$. In particular, when $S = \{(i, j) :\ a_{ij} = 0\}$, the incomplete factorization with null fill–in is known as ILU(0). To improve the quality of the factorization, many strategies for altering the pattern have been proposed. In the "level of fill–in" factorizations [8], ILU(κ), a level of fill–in is recursively attributed to each fill–in position from the levels of its parents. Then, the positions of level lower than κ are removed from S. In the experiments reported here, we



	1	2	4	5	6	8	9	8	9	10	15	20
ILU(0)	41.2	28.8	26.7	25.9	26.4	25.9	26.7	27.4	29.6	29.7	37.8	44.1
ILU(1)	26.5	22.7	23.2	24.0	26.0	27.3	30.4	33.6	34.1	36.9	47.5	61.9
ILU(2)	28.3	28.2	29.4	33.4	39.3	41.9	47.5	53.2	58.9	59.9	86.2	112.5
ILU(3)	36.4	40.7	49.3	56.8	68.0	73.0	83.0	93.1	102.9	113.1	163.3	214.3

(a) $m_\ell = \ell$.



	1	2	3	4	5	6	7	8	9
m_ℓ=1	82.3	55.5	48.4	46.1	45.6	46.2	47.3	48.5	50.1
m_ℓ=2	51.9	36.8	33.4	32.9	33.5	34.7	36.2	37.9	39.8
m_ℓ=7	30.3	23.9	23.0	23.9	25.2	27.1	28.7	30.8	32.8
$m_\ell = \ell$	26.5	22.7	23.2	24.0	26.0	27.3	30.4	33.6	34.1

(b) $\kappa = 1$.

Figure 1. Parallel Newton two–stage method, $p = 4,\ n = 125000$, Cluster 1Gb/sec.

have used these ILU(κ) factorizations for the matrices $D_k(x),\ 1 \leq k \leq p$, defined above. We have modified the SPARSKIT routine which obtains the ILU(κ) factorization in order to improve the factorizations of successive matrices with the same sparsity pattern. The stopping criterion used was $\|F(x^{(\ell)})\|_2 < 10^{-\delta}$, with $\delta = 7$. All times are reported in seconds.

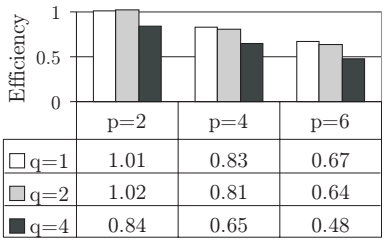
We have run our codes with problems of various sizes and different levels of fill–in for the ILU factorizations. In order to focus our discussion, we present here results obtained with $d = 50$ and $d = 72$ that lead to nonlinear systems of size $n = 125000$ and $n = 373248$, respectively.

Figure 1(a) illustrates, using four processors, the influence on the execution time of different levels of fill–in for $m_\ell = \ell$. It can be observed that the best times are obtained setting incomplete LU factorizations with levels $\kappa = 0$ or $\kappa = 1$, depending on the number of local steps $q(\ell, s, k) = q$ performed. That is, for small values of q (approximately $q \leq 5$, in this figure) the best times were obtained setting $\kappa = 1$, while for high values of q it should use level 0 of fill–in. The levels of fill–in ILU(κ) refer to the amount of fill–in allowed during the incomplete factorization. For given values of m_ℓ and q, increasing the level of fill–in κ, provides a better quality method in terms of its rate of convergence (that is, in terms of reducing the number of global iterations required), but at the cost of increasing storage. Therefore, it can be seen that, for $\kappa > 1$ this reduction of global iterations does not balance the increase of the computational cost obtained by increasing the level of fill–in.

Figure 1(b) illustrates the influence of the choice of $m_\ell,\ \ell = 0, 1, \ldots,$ in relation to the non–stationary parameter q used, for $\kappa = 1$. As it can be appreciated, the optimal choice of the values $m_\ell,\ \ell = 0, 1, \ldots$ and q, are $m_\ell = \ell$ and small values of q, respectively. However, if we use values of q higher than the optimal, the method behaves better using a constant value of linear iterations m_ℓ at each global iteration ℓ, but this optimal value is hard to predict.

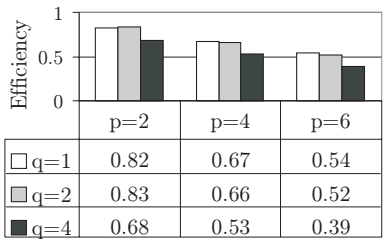
We have compared the results of these parallel methods with the well–known sequential Newton Gauss–Seidel method, the sequential Newton ILU method [10], and with respect to the same algorithm executed on a single processors. In all cases, the best sequential methods were obtained with $m_\ell = \ell$. Figure 2 illustrates the obtained efficiencies. As it can be appreciated, the parallel implementations reduce substantially the sequential times. Also, these efficiencies show a good degree of parallelism of the method treated here.

Nevertheless these algorithms need to perform the evaluation and factorization of the Jacobian at each nonlinear iteration; this step is one of the most costly. As it can be seen in Section 2, one



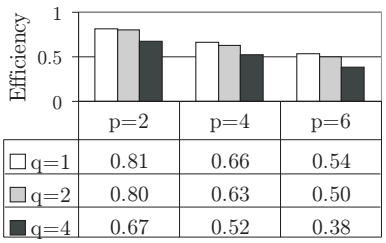
	p=2	p=4	p=6
q=1	1.01	0.83	0.67
q=2	1.02	0.81	0.64
q=4	0.84	0.65	0.48

(a) Sequential reference algorithm: Newton Gauss-Seidel.



	p=2	p=4	p=6
q=1	0.82	0.67	0.54
q=2	0.83	0.66	0.52
q=4	0.68	0.53	0.39

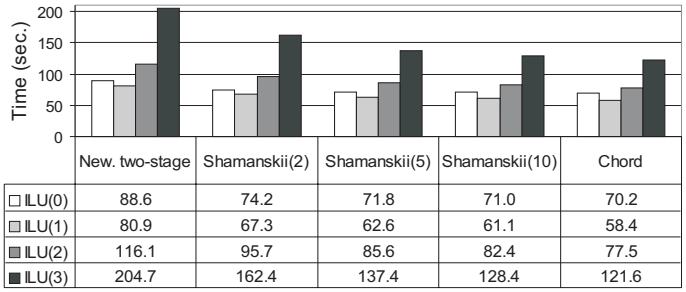
(b) Sequential reference algorithm: Newton ILU.



	p=2	p=4	p=6
q=1	0.81	0.66	0.54
q=2	0.80	0.63	0.50
q=4	0.67	0.52	0.38

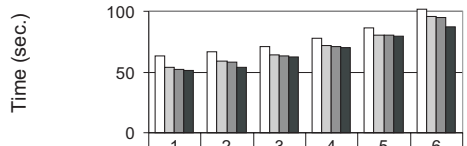
(c) Sequential reference algorithm: Newton two–stage, $p = 1$.

Figure 2. Efficiency of parallel Newton two–stage method, $m_\ell = \ell,\ \kappa = 1,\ n = 125000$, IBM RS/6000 SP.



	New. two-stage	Shamanskii(2)	Shamanskii(5)	Shamanskii(10)	Chord
ILU(0)	88.6	74.2	71.8	71.0	70.2
ILU(1)	80.9	67.3	62.6	61.1	58.4
ILU(2)	116.1	95.7	85.6	82.4	77.5
ILU(3)	204.7	162.4	137.4	128.4	121.6

(a) $m_\ell = 7,\ q = 3$.



	1	2	3	4	5	6
New. two-stage	63.2	66.8	70.7	77.4	86.5	101.3
Shamanskii(5)	53.6	59.4	64.3	71.6	80.7	95.4
Shamanskii(10)	52.4	58.5	63.4	70.7	80.1	94.8
Chord	51.5	54.0	62.7	70.0	79.5	86.8

Non-stationary parameter

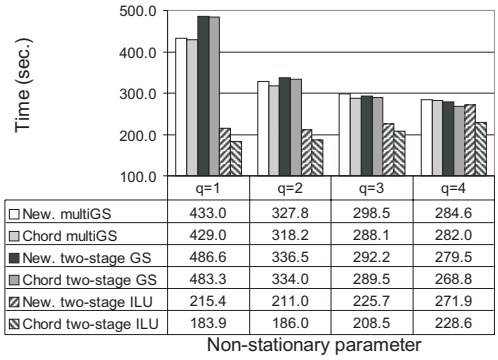
(b) $m_\ell = \ell,\ \kappa = 1$.

Figure 3. Parallel variations of Newton two–stage method, $p = 3,\ n = 125000$, IBM RS/6000 SP.

approach to reduce the cost of each nonlinear iteration of a Newton algorithm can be obtained with far fewer Jacobian evaluations or factorizations (Shamanskii method), or with only one Jacobian evaluation, before the nonlinear iteration begins (Chord method).

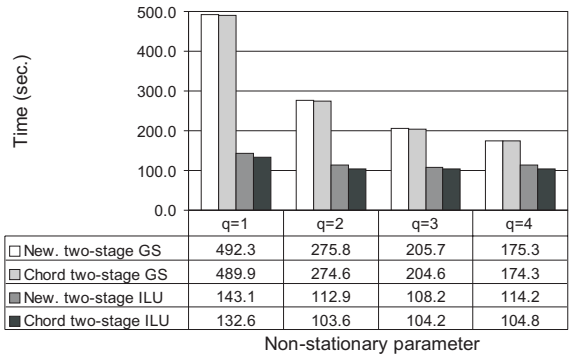
Figure 3 illustrates the behaviour of these variations of the parallel Newton method studied here. In this figure, Shamanskii(τ) indicates that the Jacobian matrix is updated each τ nonlinear iterations. The results obtained indicate that the parallel Chord methods are the best option for this problem. Moreover, the best results were obtained, again, with level 1 of fill–in (i.e., using ILU(1)).

On the other hand, we have compared the methods introduced here with both, those corresponding to the algorithms presented in [2] (denoted here by New. multiGS) and those corresponding to the algorithms presented in [1] (denoted here by New. two–stage GS). The splittings used in each case are the same as the ones used in [2] and [1], respectively. Moreover, we have implemented the corresponding Chord variation of those methods (denoted here Chord multiGS and Chord two–stage GS, respectively). The best results obtained for the methods introduced in [2] and [1], and its Chord variation, described here, have been obtained setting $m_\ell = \ell$. As it can be appreciated in Figure 4,



	q=1	q=2	q=3	q=4
New. multiGS	433.0	327.8	298.5	284.6
Chord multiGS	429.0	318.2	288.1	282.0
New. two-stage GS	486.6	336.5	292.2	279.5
Chord two-stage GS	483.3	334.0	289.5	268.8
New. two-stage ILU	215.4	211.0	225.7	271.9
Chord two-stage ILU	183.9	186.0	208.5	228.6

(a) IBM RS/6000 SP.



	q=1	q=2	q=3	q=4
New. two-stage GS	492.3	275.8	205.7	175.3
Chord two-stage GS	489.9	274.6	204.6	174.3
New. two-stage ILU	143.1	112.9	108.2	114.2
Chord two-stage ILU	132.6	103.6	104.2	104.8

(b) Cluster 1Gb/sec.

Figure 4. Comparison of parallel Newton two–stage methods, $m_\ell = \ell,\ p = 6,\ \kappa = 1,\ n = 373248$.

the numerical experiments performed with these methods indicate that the use of parallel methods using ILU factorizations to obtain the splittings is preferable to the use of parallel Newton methods using the multisplittings described in [2] and [1] (Gauss-Seidel type splittings). Concretely, we have observed a substantial reduction in the computational time when solving our model problem. By comparing the best execution times in each figure, the parallel implementations of the methods introduced in this paper provide a time reduction of about 30%-60%.

References

[1] J. Arnal, V. Migallón, and J. Penadés. Parallel Newton two–stage multisplitting iterative methods for nonlinear systems. *BIT Numerical Mathematics*, 43:849–861, 2003.

[2] J. Arnal, V. Migallón, and J. Penadés. Non–stationary parallel Newton iterative methods for nonlinear problems. *Lecture Notes in Computer Science*, 1981:380–394, 2001.

[3] B.M. Averick, R.G. Carter, J.J. More, and G. Xue. The MINPACK–2 Test Problem Collection. TR MCS–P153–0692, Mathematics and Computer Science Division. Argonne National Laboratory, 1992.

[4] A. Berman and R.J. Plemmons. *Nonnegative Matrices in the Mathematical Sciences*. Academic Press, New York, third edition, 1979. Reprinted by SIAM, Philadelphia, 1994.

[5] R. Bru, V. Migallón, J. Penadés, and D.B. Szyld. Parallel, synchronous and asynchronous two–stage multisplitting methods. *Electronic Transactions on Numerical Analysis*, 3:24–38, 1995.

[6] A. Frommer and D.B. Szyld. H–splittings and two–stage iterative methods. *Numerische Mathematik*, 63:345–356, 1992.

[7] D.A. Knoll and D.E. Keyes. Jacobian-free Newton–Krylov methods: a survey of approaches and applications. *Journal of Computational Physics*, 193:357–397, 2004.

[8] H.P. Langtangen. Conjugate gradient methods and ILU preconditioning of nonsymmetric matrix systems with arbitrary sparsity patterns. *International Journal for Numerical Methods in Fluids*, 9:213–233, 1989.

[9] A. Messaoudi. On the stability of the incomplete LU factorizations and Characterizations of H–matrices. *Numerische Mathematik*, 69:321–331, 1995.

[10] J.M. Ortega and W.C. Rheinboldt. *Iterative Solution of Nonlinear Equations in Several Variables*. Academic Press, San Diego, 1970.

[11] Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing Company, 1996.

Parallel Algorithm for Nonlinearly Unconstrained Optimization based in Parametric Trees

I. Pardines[a], D. E. Singh[b] and F. F. Rivera[c]

[a]Departamento de Arquitectura de Computadores y Automática, Universidad Complutense, 28040 Madrid, Spain

[b]Departamento de Informática, Universidad Carlos III de Madrid, 28911 Leganés, Spain

[c]Departamento de Electrónica y Computación, Universidad de Santiago de Compostela, 15782 Santiago de Compostela, Spain

Algorithms for solving unconstrained nonlinear optimization problems are computational expensive. In each iteration of these iterative methods, a search direction and a stepsize along this direction are generated. Both computations influence the number of iterations and the number of function evaluations required for the whole process. A reduction in the total number of iterations and function evaluations implies a significant improvement in the performance of the algorithm. In this context, multi-step parallel algorithms based on the execution of a sequence of trees are proposed in this paper. Different damping parameters are used to define the branches of the tree. Each branch computes a different stepsize. The performance of the parallel algorithm depends on the number of branches and the depth of the tree, which can be parameterized and controlled by the user. Results on a set of test problems to validate the efficiency of our proposals are shown.

1. Introduction

Unconstrained nonlinear optimization problems arise in many applications in science and engineering. Formally, they can be expressed as,

$$\begin{array}{ll} min & f(x): \quad I\!R^n \rightarrow I\!R, \\ x \in I\!R^n & \end{array} \tag{1}$$

where the objective function $f(x)$ is assumed to be at least twice continuously differentiable. These problems are often expensive to solve in terms of computation resources. The reason can be the size of the problem and/or the complexity of the objective function. Moreover, sometimes when $f(x)$ is complex, the gradient of $f(x)$ is not available analytically. In this cases, finite or central differences are used to approach the gradient, which implies more function evaluations. In this work, we are concerned with parallel numerical methods for solving these type of problems.

A well known solution to deal with these problems is to use quasi-Newton methods. The parallelization of these methods have been considered by several researches in the past decades. There are two different ways to deal with this problem, one is to reduce the computational time per iteration, parallelizing the most costly algebra routines, and the other is to reduce the total number of iterations of the method. The authors have been working in the context of the first approach during the last past years [10] [11]. In these papers a parallel implementation of both, the Hessian matrix update and the computation of the search direction were proposed.

Some works have been devoted to other problems where the function and the gradient evaluations require high computational times, such as a function evaluation that involves the solution of a system of differential equations or a large scale simulation. For instance, the parallel strategy to compute

the gradient at each iteration is straightforward using finite differences. Formally, this computation implies $n + 1$ function evaluations, being n the number of variables, as

$$\nabla f(x)_i = \frac{f(x + h_i e_i) - f(x)}{h_i}, \tag{2}$$

where h_i is a small stepsize, and e_i denotes the i^{th} unit vector. In this case, if P processors are available, each processor can compute $\lceil n/P \rceil$ evaluations of the function, but during the evaluation of $f(x)$ in the line search, $P - 1$ processors are idle. To avoid this drawback, Schnabel proposed, in [13], a speculative technique to evaluate $max\{P-1, n\}$ components of the gradient at the updated solution point at the same time that this solution point is computed. Furthermore, Byrd, Schnabel and Shultz [2] suggest to speculatively compute the gradient and some portion of the Hessian matrix, and to incorporate this partial Hessian information into the BFGS update.

Finally, there is other ways to approach the parallelization of quasi-Newton methods that has been suggested by Lootsma [7], van Laarhoven [6], and more recently, by Phua [12]. The idea is to compute, at each iteration, P independent search directions (multi-search methods) or P different steplengths along a given search direction (multi-step methods). For massive parallel systems these two approaches can be combined to achieve high performance. The computational time is reduced by looking for different solution points at each iteration. There are two reasons for this reduction; first, the decrease in the number of function evaluations in the line search procedure and, second, a fast convergence of the method.

In this paper, a description of quasi-Newton methods, specifically the BFGS method, is presented in Section 2. Our proposals based on a parametric tree to parallelize a multi-step BFGS method are described in Section 3. Results obtained by our parallel methods in a cluster of PCs are shown in Section 4. Finally, Section 5 summarizes the main conclusions of this work.

2. Quasi-Newton methods

The success of Newton-type methods is the use of the curvature information provided by the Hessian matrix, which allows a quadratic model of the objective function. However, quasi-Newton methods are based on the idea that an approximation to the curvature of a nonlinear function can be computed without explicitly forming the Hessian matrix. This information is collected as the iterations of a descent method proceed, using the behavior of the objective function and its gradient. Therefore, these methods use an approximation to the Hessian matrix instead of the Hessian itself. At each iteration of the method, when a new solution point is considered, the approximation of the Hessian matrix has to be updated. One important feature of quasi-Newton methods is the choice of the Hessian matrix approximation. There are different update formulas [5]. One of the most commonly used to solve unconstrained nonlinear optimization problems is the Broyden-Fletcher-Goldfarb-Shanno (BFGS) update.

The BFGS method works with the Hessian matrix approximation, however, to solve the required system of equations is more efficient to use the inverse of the Hessian matrix approximation. The algorithm that uses this matrix is called the inverse BFGS method, and it is the one that has been considered in this work.

Given an initial point x_k, a gradient g_k and a Hessian matrix approximation H_k, the inverse BFGS method executes a set of stages at the k-th iteration. Firstly, a direction of search $d_k = -H_k^{-1} g_k$ is computed. After that, an optimal stepsize α_k, along this search direction from x_k is found. Therefore, the next iteration point is given by $x_{k+1} = x_k + \alpha_k d_k$. Finally, the approximated Hessian matrix is updated by the correction matrix D_k, in such a way that $H_{k+1} = H_k + D_k$.

A detailed profiling analysis shows that one of the most expensive stages of the inverse BFGS method is the computation of the steplength along a search direction in every iteration. Both, the steplength and the search direction influence the number of iterations required for the whole optimization process. The search direction can be modified using a different quasi-Newton update. However, as the BFGS update presents the best behavior in most cases, we just propose to modify the computation of the steplength. The line search procedure is summarized in detail in the next section.

2.1. The line search procedure

The line search procedure computes the steplength at each iteration. Computing the steplength α_k at the k-th iteration implies the solution of the following one-dimensional optimization problem [8]:

$$f(x_k + \alpha_k d_k) = min \quad f(x_k + \theta d_k), \quad with \quad 0 < \theta \leq \alpha_{max}. \tag{3}$$

The selected steplength has to verify the Wolfe's conditions,

$$f(x_k + \alpha_k d_k) - f(x_k) \leq \mu g_k^T d_k, \tag{4}$$

$$\nabla f(x_k + \alpha_k d_k) \geq \eta g_k^T d_k, \tag{5}$$

being μ, $\eta \in (0, 1)$.

The best methods to solve this type of problems are the so called safeguarded procedures. These methods can be considered combinations of the bisection and the linear interpolation methods. Their efficiency is based on the use of information from the objective function. Polynomials are frequently used to interpolate $f(x_k + \theta d_k)$. If the approximation is inaccurate the procedure may diverge, so an interval of uncertainty $[a, b]$ is maintained as safeguard.

The line search procedure finds a sequence of improving estimates of a minimizer of $f(x_k+\theta d_k)$ in the interval $(0, \alpha_{max}]$. In our proposal, safeguarded cubic interpolation has been used to compute the sequence of estimates. The procedure computes an interval of uncertainty and the two "best" points of the objective function are obtained to be used as interpolation points. Each time an estimate is computed, for being selected it has to verify the Wolfe's conditions. If these conditions fail the interval is changed and a new estimate is searched.

The selection of the initial step is critical to achieve a global convergence of the method. At the k-th iteration, in the first evaluation of function f, a potentially small steplength $\alpha_{k0} = min(1, \gamma)$, where $\gamma = r(1 + ||x_k||_1)/||d_k||_1$, is used. The parameter r, called damping parameter, limits the change in the current solution point x_k during the line search. Therefore, evaluations of the objective function at meaningless points are prevented. A suitable damping parameter can be fixed according to the particular features of the problem. For example, low values, such as 0.1 or 0.01, may be helpful when the function varies rapidly.

We are interested in multi-step methods that compute, at every iteration, several steplengths. We suggest to initialize different values of α using different damping parameters, and establishing several parallel quasi-Newton executions, each one starting from a different initial step. The parallel strategies are described in the next section.

3. Parallel tree techniques

A set of parallel implementations of a multi-step quasi-Newton method based on the parallel execution of a tree structure is proposed. Each branch of the tree is characterized by a different selection of the damping parameter at each node. A node is related with a different iteration of the

method, so each branch runs a quasi-Newton method in a processor of the parallel system with a different sequence of damping parameters. The way in which the damping parameters are defined, and the criterium to select the initial branches of the next tree, each time a checkup is made, are the differences between the parallel methods. Following, a description of these strategies is presented, highlighting their benefits and drawbacks.

3.1. Method 1

Since this method is based on the execution of the computations associated to a tree structure, it is necessary to describe how that tree is built. The number of nodes per branch (β) and the number of branches characterize the tree. β is an externally fixed parameter, and the number of branches of the tree corresponds to the number of damping parameters $n(r)$. Each branch consists of a set of nodes each one with the same value of the damping parameter. This implementation of the tree is justified by the empirical observation that the best solution is achieved selecting the same damping values during a certain number of concurrent iterations. In the example of Figure 1(a), five values of r are considered and $\beta = 2$. Therefore, five branches are generated after pairs of iterations.

Once the tree has been defined, in the first iteration, the algorithm starts the execution of the computational load associated to this tree. Every branch runs a quasi-Newton method in a different processor with the associated values of damping. Therefore, as many processors as the number of branches are needed. This tree remains active for β iterations. As soon as the algorithm completes the computations associated to the tree, the branch that gives the best solution at the moment is checked. The criterium to select the solution is based on the minimum value of the objective function. Each time a checkup is performed to select the best branch, the current solution point associated to this branch has to be broadcasted to the rest of the processors of the system. Therefore, a message is needed after each β iterations. The arrows in Figure 1 represent *Allreduce* operations to decide which branch achieve the best solution.

When the processors receive the current solution point, a new tree is performed, starting from this point. The process continues until the quasi-Newton method converges.

Our heuristic presents two degrees of freedom: the width and the depth of the tree. The number of branches is characterized by the width of the tree, and it depends on the amount of different damping parameters considered. The depth of the tree is defined as the number of iterations between checkups. The performance of the method is dependent on the parameters that define the tree. However, in practice the behavior of the method is unpredictable, because the method presents a drawback. That is, sometimes the branch that is selected as the best in a certain iteration is not the branch that converges in the minimum number of iterations. For this reason, the method do not always achieve the best solution. So, a second method is proposed as an alternative.

3.2. Method 2

This method is used to deal with the main drawback of Method 1. The idea is to select more than one branch each time the checkup is performed. Therefore the m best branches are kept alive. The value of m depends on the number of processors of the parallel system. In this way, after the execution of the first tree, m solution points of the quasi-Newton method are available. These points are used as starting points to perform the branches of new m trees. These trees have again as many branches as damping parameters are used, as can be seen in Figure 1(b). The process continues until the solution of the optimization problem have been achieved.

The disadvantage of this method respect to Method 1 is that more computations are needed. In this case, $P = m \cdot n(r)$ processors are required, instead of the $n(r)$ processors of Method 1. Note that, from a parallel point of view, in the computation of the first tree $(m-1) \cdot n(r)$ processors are

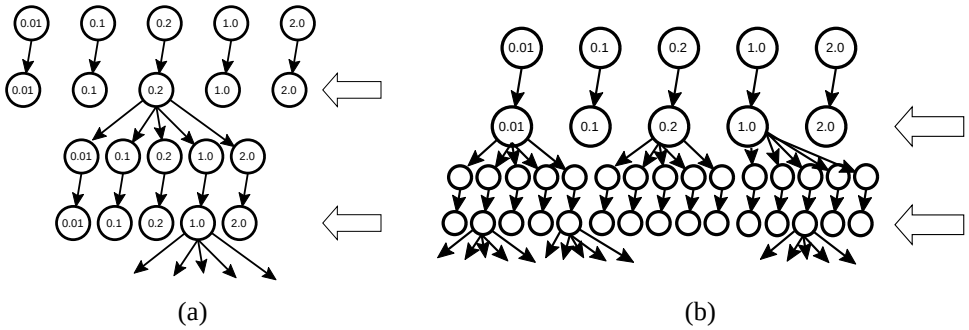


Figure 1. (a) An example of the application of Method 1. (b) An example of application of Method 2 with $m = 3$. For both cases the set of dampings=$\{0.01, 0.1, 0.2, 1.0, 2.0\}$ and $\beta = 2$.

idle.

3.3. Method 3

Experimental results show that the selection of damping parameters is important in order to achieve high performance in the execution of the parallel algorithm. The previous two methods use a set of unchangeable damping parameters given by the user. However, a new degree of freedom can be added, if the values of the damping parameters are under the user control. Depending on the type of the problem, the user can decide to use a set of random damping parameters when the behavior of the problem is unknown (Method 1 and Method 2) or to start with a suitable damping and try to find a better value. In this last case, we propose to modify the chosen damping a certain percentage x. If the number of variations over this damping value is v, a tree with $2v + 1$ branches is generated. The proposed damping parameters for these branches are:

$$r_i - v \cdot x\%, \cdots, r_i - x\%, r_i, r_i + x\%, \cdots, r_i + v \cdot x\% \tag{6}$$

Therefore, the number of processors in this method is $P = m \cdot (2v + 1)$. The user decides, based on the size of the parallel system, to increase m or v. In both cases, the controlled parameter is the width of the tree.

Method 3 can be used after an initial fast execution of Method 1. In this way, Method 1 can be executed with a selected set of dampings during some few iterations, then, the damping parameter that lead to the best solution in Method 1 is used as the starting damping for Method 3. This method will refine the damping parameter by controlled variations, improving the performance.

4. Experimental results

The parallel strategies were implemented using MPI in a cluster of PCs. Our benchmark consists of three well known optimization problems from the CUTE library [1] called “Power”, “Penalty1” and “Powellsg”, that come from three different and representative applications. The results are compared with the solutions achieved by the optimization software MINOS [9]. The percentage of reduction in the total number of iterations ($Iter$) and in the number of function evaluations ($Efun$) achieved by Method 1 respect to MINOS are shown in Figure 2. A set of five different damping parameters, and values of β from 2 to 10 are used to solve these problems in a parallel system of 5 processors.

The efficiency of Method 1 depends on the type of the problem. The highest performance is achieved for the ”Power” problem, where a maximum speedup of 4.15 is obtained. For the other two problems low values of β implies discrete results. However, as the depth of the tree increases, the performance improves. Note that, even a small reduction in the number of function evaluations is important, because it implies a minor cost in the computation of the steplength. Both reductions are more significant when there is no analytic gradient available, and it has to be computed by finite differences.

It can be observed that for some depths of the tree, poor improvements are achieved. The reason is that according to the mentioned drawback of Method 1, in some cases the best branch selected in each iteration is not the one that produces a quicker convergence.

The same benchmark and the same set of damping parameters have been used to study Method 2. In this case, the $m = 10$ branches with the lowest value of the objective function are selected each checkup. So, the number of processors required in this case is $P = 50$.The results of Method 2 versus Method 1 are shown in Figure 3 for problems with $n = 20$. Results for the “Power” problem are not shown because they are the same for both methods. The reason is that this is a well conditioned problem for which the branch selected as the best one in Method 1 leads always to a fast convergence. For the other problems, the greater improvement is achieved in the reduction of the number of iterations, and specifically, in the “Powellsg” problem, which has the worst results with Method 1. Also, it is interesting to highlight that for some depths of the tree the solution achieved by Method 1 fits with the solution of Method 2.

A comparative of the three methods is presented in Figure 4. Method 3 has been applied keeping only one branch alive each checkup in order to use less processors, explicitly $P = 7$ is used. The damping parameter that leads to the best value of the objective function in Method 1 is used. The selection of the parameters x and v has been made by means of empirical experiments.

In this figure, the different behavior of the two problems is stood out. From the ‘’Penalty1” problem, improvements between 40% and 50% in both the number of iterations and the number of function evaluations are achieved. Similar results were obtained for the three methods with values of β over 4 iterations. Method 3 shows a good behavior, very close to Method 2. Therefore, as the number of processors needed by this method is much smaller than the one used in Method 2, Method 3 seems the best option to optimize problem “Penalty1”. The behavior with the depth of the tree is much irregular for the problem “Powellsg”. In this case, Method 2 presents the best

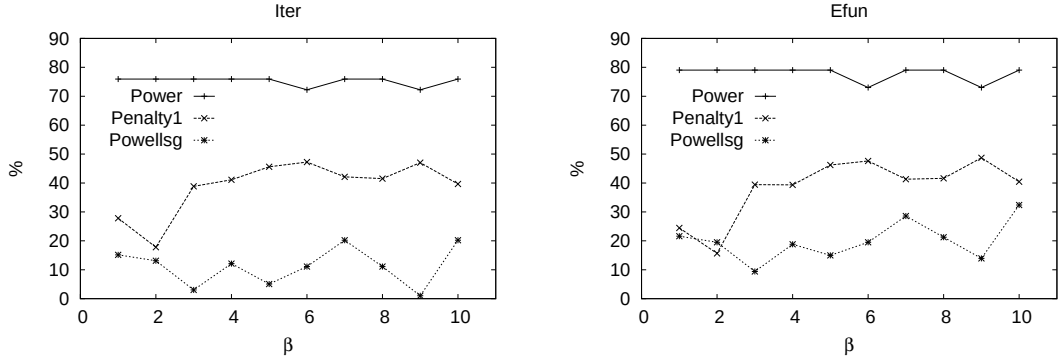


Figure 2. Percentage of improvement obtained by Method 1 for problems with 20 variables in a parallel system of 5 processors.

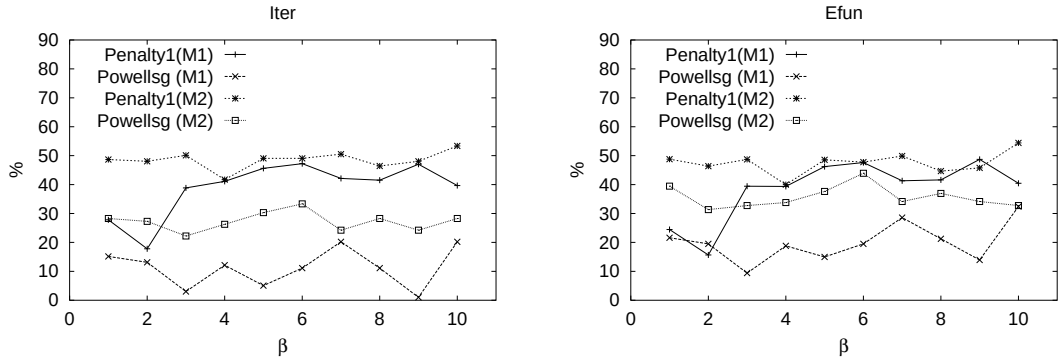


Figure 3. Improvement obtained by Method 2 versus Method 1 for problems of size $n = 20$. The number of processors is P=5 in Method 1 and P=50 in Method 2.

performance in most cases. However, when the depth of the tree is equal to 4, Method 3 achieved the best improvement, over 40%. This example stands out how the suitable depth of the tree depends on the type of the problem and on the method.

5. Conclusions

In this work different strategies to parallelize quasi-Newton methods are described. Three parallel multi-step algorithms based on the use of parametric trees are proposed. Our objective is to improve the performance of quasi-Newton methods by reducing the number of iterations and the number of function evaluations. The proposed heuristics present two degrees of freedom, the width and the depth of the tree. The width is related to the number of damping parameters, and the depth is defined as the number of iterations between consecutive checkups. Method 3 adds the set of damping parameters as a new way of controlling the width of the tree.

Depending on the type of the optimization problem, the best solution is obtained for a tree with a certain width and depth. However, we can conclude that choosing a suitable set of damping parameters and an appropriate depth of the tree, an improvement in the performance of the quasi-Newton methods is obtained. Moreover, Method 3 turns out as the best alternative when the behavior of the problem is known, since achieves good performances using small parallel systems.

Acknowledgments

This work has been supported by CICYT under projects TIC 2002/750, TIN 2004-02156 and TIN 2004-07797-CO2.

References

[1] I. Bongartz, A. R. Conn, N. Gould and P. L. Toint, CUTE: Constrained and unconstrained testing environment, ACM Transactions on Mathematical Software 21 1 (1995) 123-160.

[2] R. H. Byrd, R. B. Schnabel and G. A. Shultz, Parallel Quasi-Newton methods for unconstrained optimization, Mathematical Programming 42 (1988) 273-306.

[3] J. A. Ford and I.A. Moghrabi, Minimum Curvature Multistep Quasi-Newton Methods, Computers and Mathematics with Applications 31 (1996) 179-186.

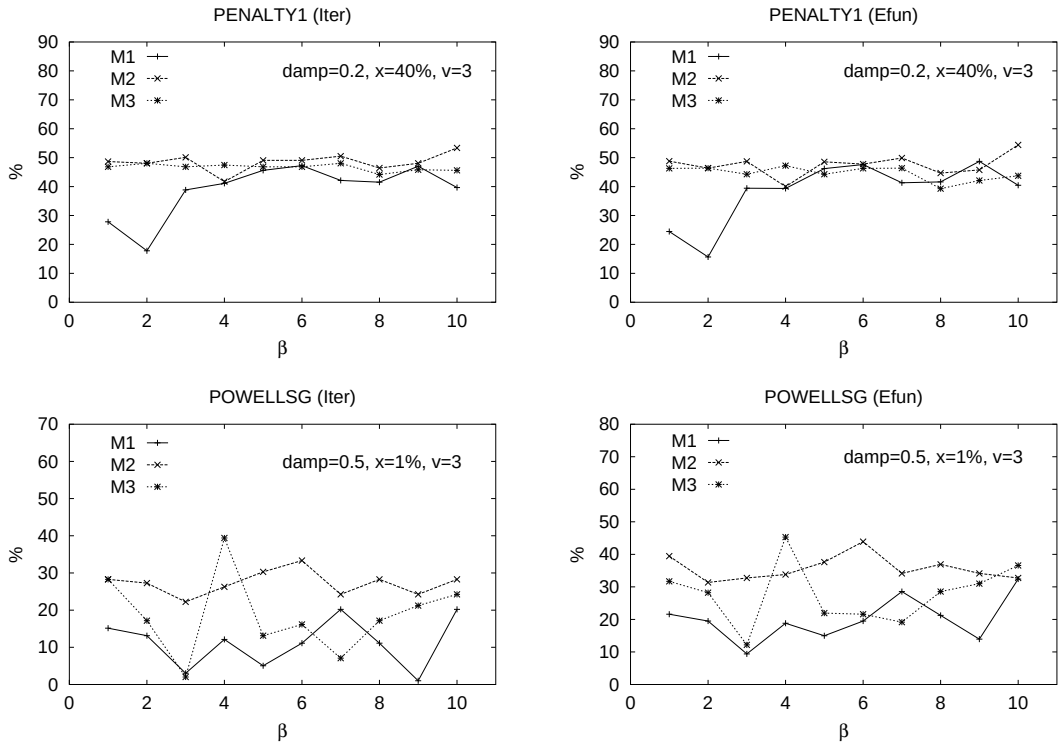


Figure 4. Comparative of the three methods for problems of size $n = 20$.The number of processors in Method 3 is P=7.

[4] J. A. Ford, Implicit Updates in Multi-step Quasi-Newton Methods, Computers and Mathematics with Applications 42 (2001) 1083-1091.

[5] Philip E. Gill, Walter Murray and Margaret H. Wright, Practical Optimization (Academic Press, London, 1981).

[6] P. J. M. van Laarhoven, Parallel variable metric methods for unconstrained optimization, Mathematical Programming 33 (1985) 68-81.

[7] F. A. Lootsma and K. M. Ragsdell, State-of-the-art in parallel nonlinear optimization, Parallel Computing 6 (1988) 133-155.

[8] Jorge J. Mor and David J. Thuente, Line Search Algorithms with Guaranteed Sufficient Decrease, ACM Transactions on Mathematical Software 20 3 (1994) 286-307.

[9] Bruce A. Murtagh and Michael A. Saunders, Minos 5.4 User's Guide, Technical Report SOL 83-20R, Department of Operations Research, Stanford University, 1983.

[10] I. Pardines, J. J. Pombo and F. F. Rivera, Parallel Algorithm for Backward and Forward Sweeps of Plane Rotations, in: Proc. IASTED International Conference Applied Informatics (Acta Press, Zurich, 2001) 418-423.

[11] I. Pardines and F. F. Rivera, Parallel Quasi-Newton Optimization on Distributed Memory Multiprocessors, in: Parallel Computing, Advances and Current Issues (Imperial College Press, London, 2002) 338-345.

[12] K. H. Phua and R. Setiono, Multi-step, multi-directional parallel algorithms for unconstrained optimization, in: Optimization Techniques and Applications (World Scientific, Singapore, 1992) 481-487.

[13] R. B. Schnabel, Concurrent function evaluations in local and global optimization, Computer Methods in Applied Mechanics and Engineering 64 (1987) 537-552.

Parallel Global Optimisation for Oil Reservoir Modelling *

Susana Gomez[a], Nelson del Castillo[a]

[a] IIMAS, Univ. Nacional A. de Mexico, Mexico, susanag@servidor.unam.mx

1. Introduction

The objective of this work is to develop a robust mathematical and computational tool to forecast the production of an oil reservoir at a reasonable computational cost. To forecast production it is neccessary to simulate the flux in a porous media, to obtain the pressure and saturation (state variables), time and space dependent. This implies the solution of a highly complex system of PDEs, which has been thoroughly studied. There are today, a set of commercial simulators that incorporate the latest numerical efficient and robust methods, that guarantee the existence and uniqueness of the solution and the desired precision. Here we will use one of this simulators, called ECLIPSE, able to handle a wide espectrum of reservoir conditions (3-D, multiphase flow, compositional, etc).

This system depends linearly or non-linearly, on a set of coefficients or parameters p, that are space dependent, related to the porous media characteristics (transmissibility and porosity) of the site under study. In practice, these parameters are not known, although some geological and geophysical information might be avalilable. However, the high sensitivity of the simulation and forecast (simulation for future periods of time) to the knowledge of these characteristics p, makes their determination the main and most difficult problem. To be able to find these coefficients, using limited historical data on the state variables measured at the wells, a data-fitting least-squares optimisation problem can be solved [9]. This inverse problem is generally known as History Matching characterisation.

Although the solution of the flux equations is well-posed in the Hadamard sense, the inverse parameter identification problem is not. This implies that uniqueness of the optimal solution is not guaranteed unless more a-priori information on the solution is known and incorporated in the problem definition. The continuity of the solution to the data is neither satisfied here (the inverse operator may not be continuous), as a small error in the measured data, and/or in the numerical solution of the direct problem, can produce optimal solutions completely misleading (see for instance [12,10]). For this continuity problem, some regularisation method has to be used to be able to get the best possible stable approximation to the solution. Also, the evaluation of the objective function (at least once at each iteration of the optimization process), implies the solution of the PDEs system, making the History Matching a highly computationally intensive problem.

Then, considering the non-convexity of the objective function, and taking into account the non-continuity problem, the goal here is to get a set of stable optimal solutions with good match to the data, in reasonable computer time. Thus, it is neccessary to develop a robust, reliable and efficient global optimisation method. We then solve the flux equations for each optimal solution (a set of coefficients) at future time periods, to produce a set of scenarios of production forecast.

Here, we work with the Parallel Tunneling Global Optimisation Method (PTGOM) [8], that finds a set of local optima with good match to the data. This method requires the use of a gradient based local optimisation method, and thus its speed and robustness depend partly on the effectiveness of the local method. In this work, we also test the ability of the local method used, the Truncated Gauss Newton with Conjugate Gradient as the linear solver, to regularise the optimal local search. In a

*This work has been partially supported by UNAM-PAPIIT, GRANT No. IN-112999, CONACyT GRANT No.35458-A and by Consortium OPTIMA (Total and Agip).

future publication we give comparison results when a Limited Memory Quasi-Newton method is also used.

Although PTGOM is deterministic in its general design, in the present version it makes a space search starting from random directions, to locate points in other valleys and this gives the possibility to parallelise them. Other stochastic methods have been used to solve the history matching problem, but their computational cost is still an open issue. Numerical results will be given here for a benchmark problem called PUNQ-S3.

2. Global Optimisation

In order to introduce notation, we will call $Mod(p)$ the solution of the flux system of PDEs, for a given value of the porous media characteristics or parameters $p = g(x, y, z)$. The state variables, which are space and time dependent $sv(x, y, z, t)$, are the bottom-hole presure BHP, the water-cut WCT and the gas-oil ratio GOR.

To find p, we have to solve an inverse problem: Given a limited set of measurements $data_{i,j,k}$ of the state variables sv mentioned above, measured at the observation wells at several time intervals (where i is the well number, j is the measurement number at each well and k is the time interval), find the parameters p that produce $Mod(p) \approx data_{i,j,k}$ as accurately as possible.

This inverse problem (given the effect sv find the cause p), can be posed as a least-squares data fitting optimisation problem,

$$\min \qquad F(p) = \frac{1}{2} \parallel f(p) \parallel_2^2, \quad f(p) = Mod(p,t) - data_{i,j,k}, \tag{1}$$

where $f : \Re^n \rightarrow \Re^m$, and $f(p)$ is the residual vector. Some a-priori information on the solution, based on geological knowledge of the site under study, allow us to impose some bound constraints on the parameter values

$$p_{min} \leq p \leq p_{max}. \tag{2}$$

In order to get a set of stable local optimal solutions with good match to the data, a global optimisation method has to be used, with certain characteristics that most well known methods fail to have. A more detailed description of these characteristics will be given in a future publication.

In this work, we will use the Tunneling Methods and exploit their stochastic element to design a parallel version. A brief description of these methods is given now, and the reader is referred to [15,14,1].

2.1. The Sequential Tunneling Method

The method was designed to find the global minimum x_G^* of a general function $F \in C^2$, with bound constraints and/or general non-linear constraints [14].

The basic deterministic idea of the Method is to find a local minimum x^* with $F(x^*) = F^*$ during a minimisation phase, using a local bounded optimisation method (gradient based), and then during a tunneling phase, find a point x_{tu} in another valley with $F(x_{tu}) \leq F^*$. This inequality implies that valleys whose minimum is above the level F^* will not be found. Once in another valley, starting from x_{tu}^*, it finds again another local minimum, ending with a sequence of local optimal solutions with decreasing function values $F(x_1^*) \geq F(x_2^*) \geq ... \geq F(x_G^*)$, and ignoring all the local minima with larger values than the best already found. These two phases are repeated alternatively until convergence to the global solution or until a prefixed number of function evaluations or CPU time, is achieved.

2.1.1. Tunneling Phase

Once a local minimum has been obtained, to be able to tunnel from one valley to another using gradient-type methods, it is necessary first to destroy the minimum, placing a pole at the last minimum point x^* in such a way that at $x = x^*$ the transformed function $T(x)$ is infinite and that $T(x) \leq 0$ at points $x \neq x^*$ whenever $F(x) \leq F^*$. To place a pole that satisifes these conditions we use here the Exponential Tunneling Function [1].

$$T(x) = (F(x) - F^*) \exp\left(\frac{\lambda^*}{\parallel x - x^* \parallel}\right) \leq 0. \tag{3}$$

An alternative Tunneling function (that we call classical) can be found in [15,14]. Parameter λ^* is the strength of the pole and plays an important role. The tunneling function $T(x)$ inherits the multimodality of the original function $F(x)$, and the local method used to solve problem 3, may find a critical point of $T(x)$. If λ is taken large enough, a smoothing effect takes place (removing critical points of $T(x)$) although in practice this large value of λ, that depends on the first and second derivatives of F, cannot be computed explicitly. Also, the larger λ is, the wider the pole becomes, and a local valley within that neiborhood of x^* might not be found. The method then, would take the smallest λ that gives a descent direction for $T(x)$ and the use of mobile poles to deal with the critical points of $T(x)$, allows the local method to find the solution of problem 3. A detailed description on the implementation of the method and the way to handle mobile poles and relative tolerances, can be found in [4].

To solve inequality problem 3, descent directions are generated with the same local algorithm used in the minimisation phase, with appropriate stopping conditions to check convergence for problem 3.

This method has also some convenient properties not shared by most other global methods: it is able to find minima at the same level, with equality satisfied in 3. But in order to avoid finding again minima already found at the same level, the poles at the minima already found have to be active, and the Tunneling function has to be modified in the following way

$$T(x) = (F(x) - F^*) \prod_{i=1}^{l} \exp\left(\frac{\lambda_i^*}{\parallel x - x_i^* \parallel}\right) \leq 0. \tag{4}$$

In applications like the one we solve in this work, where a set of minima with small residuals is sought, the so called minima at the same level can be defined to a certain tolerance and thus the searched points in other valleys would satisify the modified inequality $F(x_{tu}) \leq F^* + TOL$. This tolerance may be given by the expected value of the residual $f(x)$.

2.2. Parallel Design

To start the seach for x_{tu} that satisfies $F(x_{tu}) \leq F^*$ through 3, we need to take an initial point x^0 in a neighborhood of the last local minimum found x_i^*. In this work we have taken this initial point along random directions

$$x^0 = x_i^* + \epsilon r$$

where ϵ depends on the scale and the needed precision of the problem (see [4]), and r is a random direction within $r \in [-1, 1]$. From this initial point the search for the solution of inequality problem 3 starts, using the same method we used to solve the minimisation problem to generate descent directions and step lengths. If the search from a prefixed number of initial points (in different directions)

in the neighborhood of x_i^* has been unsuccessful, the search continues starting from points taken in the whole feasible region until a maximum prefixed number is reached.

The efficiency of the search for points in other valleys after a new minimum has been found, depends strongly on which direction r the initial point $x \neq x^*$ for the tunneling phase is taken. Thus the idea that supports the parallelisation of the method, is to allow each processor to start the search in a different direction. This is specially important when the number of unknown parameters or when the number of local optima are large, as the search in the subspace defined by the level set $f(x) \leq f^* + TOL$, has to be exhaustive.

However, in order to make the parallel process efficient, avoiding too many message passing, each processor will carry out independently both phases: the tunneling phase to search for points in other valleys, and if no one else has found a better minimum, once the point x_{tu} is found, it continues the search for the local minimum of that valley, by performing the minimisation phase.

A master processor has to control the process, so it finds the first local minimum starting from a given initial set of parameters, obtained using geological information of the site and sends this first minimum to all other processors. Then, it receives the new minima found by any processor, checks that it is different that the ones already reported, in which case it sends it to all other processors. It also checks for the stopping conditions, that in this application means the CPU time or a fixed number of local minima to be found determined a-priori. It sends a message to all processors to finish the run, generates the output files and stops.

The slave processors, will check for new messages concerning new minima, only while performing a tunneling phase, that will continue now from the new minimum received, creating a pole to destroy it and updating the function value F^*. This phase proceeds now to find a point x_{tu} with $F(x_{tu}) \leq F^* + TOL$. While in the minimisation phase, the slave processor only checks the general stopping messages.

2.3. Local optimisation, Regulatisation and Scaling

In this work the local optimisation method we are using is a modified version of the code TRON (see [16]) based on a Truncated Newton Method with a trust region strategy for global convergence. Here we test the Gauss-Newton approximation $H = J^T J$ to the Hessian, where J the Jacobian matrix is given by the simulator. The Gauss-Newton system of linear equations, used to find a descent direction s_k at the non-linear iteration k, is solved using a Conjugate Gradient Method.

During the History-Matching, the Hessians may be ill-conditioned due to the ill-posednes of the parameter estimation inverse problem. One of the consequences is that the valleys of the non-convex objective function may become very flat, and convergence to the local optimal solutions may not be achieved with enough precision. Scaling by variable transformation converts the variables from units that typically reflect the physical nature of the problem to units that display certain desirable properties during the minimisation process. For global optimisation, the shape of the valleys of the non-convex objective function can be modified through scaling [11], as although the objective function values would not change, the gradient and Hessian of the problem will be transformed. If the minimisation problem has simple bounds on the variables, as is the case here, say $x_i \in [l_i, u_i]$ for $i = 1, ...n$, the new scaled variables $y_i \in [l_i^{new}, u_i^{new}]$ can be defined as

$$y_i = \left(\frac{u_i^{new} - l_i^{new}}{u_i - l_i} \right) x_i + \frac{u_i l_i^{new} - u_i^{new} l_i}{u_i - l_i} u_i - l_i \tag{5}$$

This transformation [7] can be written in matrix form as $x = Dy + c$ where D is a diagonal matrix and c is a vector. Although the function in the scaled variables $h(y)$ is not altered $h(y) = F(x)$, the

derivatives of the objective function are scaled. Let g_y and H_y denote the gradient vector and the Hessian matrix of the scaled variables, and g and H the original ones. These derivatives are then related by

$$g_y = Dg \qquad \text{and} \qquad G_y = DGD$$

Hence, even a "mild" scaling may have a substantial effect on the Hessian, and this in turn may significantly alter the convergence rate of the optimisation algorithm. The linear system becomes preconditioned by matrix D.

Due to the non-continuity of the solution to the data, a small perturbation of the data (measurements errors), can cause a large perturbation of the solution. To prevent this error propagation, the degree of approximation to the solution has to be limited and regularisation methods have to be used to balance the degree of approximation (the regularisation parameter) and the error propagation. Here, we study the regularisation properties of iterative methods [13], by stopping the Conjugate Gradient iteration used to solve the Gauss-Newton linear system that finds the descent direction. As the Hessian approximation $H = J^T J$ may be ill-conditioned, it is necessary to stop the iteration before the error of the data propagates. We use here the triangle method (see [5], where it was applied for linear systems with errors) to find the corner of the L-curve generated with the Conjugate Gradient iterations, to determine automatically the stopping iteration (which becomes the regularisation parameter). This parameter re-defines the size of the trust region for the next Gauss-Newton iteration. Also, as the scaling described before implies a diagonal preconditioner of the Gauss-Newton system, it may have a regularisation effect that will also be tested here.

3. Application to History Matching of Oil Reservoirs

The PUNQ-S3 problem [17] has recently become quite popular as a sort of benchmark for history matching and risk-analysis methodologies. It is a dynamical reservoir model based on a real west Africa field, which has been discretised using a 19x28x5 corner-point grid, with 1,721 active cells. Strong water supports come from north and west, while two faults close the reservoir at east and south, and a small gas cap is present at the top of the formation. A history period, simulating 8 years of production from six wells located close to the gas-oil contact (GOC), was generated by The Nederland Organisation for Scientific Research (TNO) using geostatistical distributions of porosities and permeabilities. Gaussian noises have been added to the collected well data to reproduce a real measurement process. Then, 8 years of forecast with five additional infilling wells have been simulated. The data set, consisting of noisy well-data, grid structure, transmissibility and porosity distributions, is available at the TNO web site.

The complexity of the parameter identification problem was avoided in this work by adopting a set of parameters, based on Gradzone Analysis [2], obtained by the TOTAL Geoscience Research Centre (GRC). In this work, we used the aforementioned gradzones in which the available a priori geological information was also included in the analysis [3]. For PUNQ-S3, this analysis leads to a vector of 10 history matching parameters p, 5 porosities and 5 transmissibilities , which contains a signature of the geological model. One multiplier for each property was assigned to every layer. To restrict the evolution of the system into a physically reasonable region, simple bounds, acting as perfectly absorbing surfaces, are imposed to the parameters as follows $0.1 \leq p_i \leq 3, \quad i = 1, ...10.$

4. Numerical Results

All the experiments were run on a Beowulf cluster with 8 nodes with two pentium III 1 Ghz at each node. Communication is performed with ethernet at 100Mbps, and Linux system. In this section we show a limited set of results and a complete discussion and graphs will be given in a future publication. In Figures 1 and 2, the different scenarios obtained with each local minima (min1, min2 etc.) with good match to the data are shown for two different wells, over 16 years. The first 8 years show the history matching and the next 8 are the forecast results. In the graphs, the real data is referred as *data*. In Figures 3, 4 and 5, the same set of scenarios is presented for the whole field, for water, gas and oil. In Figure 6, the important effect of scaling (sca) and regularizing (ilcu)is shown. In Figure 7 the graph shows the computer time behaviour when 1,3,5 and 7 processors were used. In Table 1 we show the results comparing the sequential versus the parallel versions. The latest is 17 times faster. Finally in table 2 the speedup and the efficiency of the parallel implementation is given.

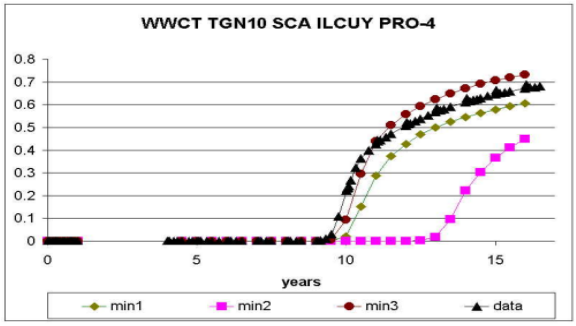


Figure 1. Scenarios produced by 3 different minima and the real data, at Well 4

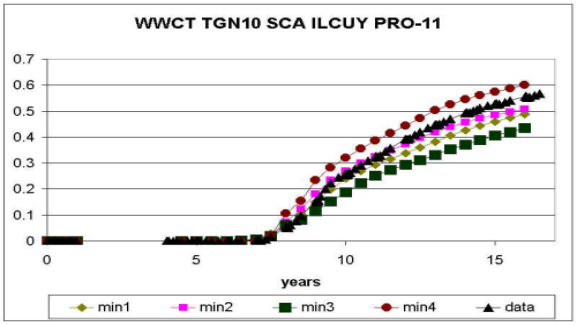


Figure 2. Scenarios produced by 4 different minima and the real data, at Well 11

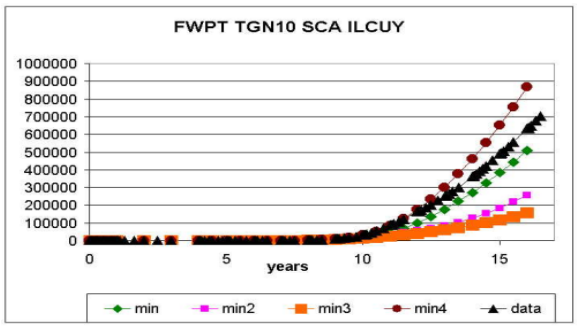


Figure 3. Field H-M and Forecast.Water

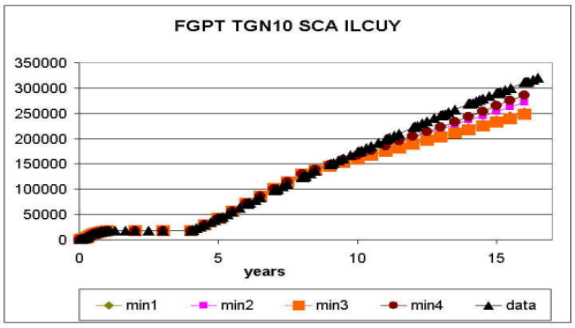


Figure 4. Field H-M and Forecast. Gas

	F^*	No. Local Min	No. Function Eval.	Time(hours)
Tun-TGN-Seq	0.2884	3	1026	17:35
Tun-TGN-Par	0.2734	4	67	1:10

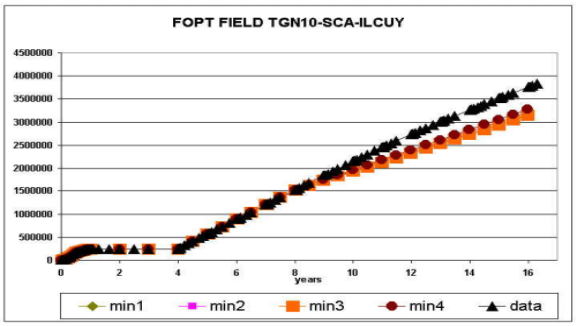


Figure 5. Field H-M and Forecast.Oil

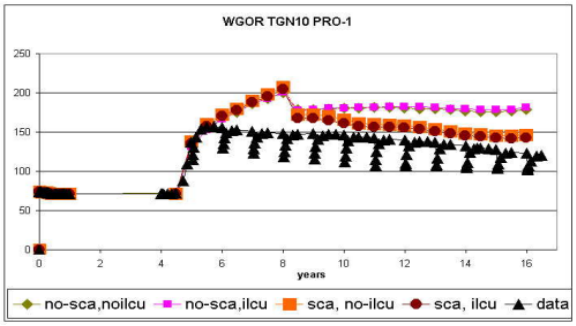


Figure 6. Good effect of Scaling (sca) and Regularising (ilcu)

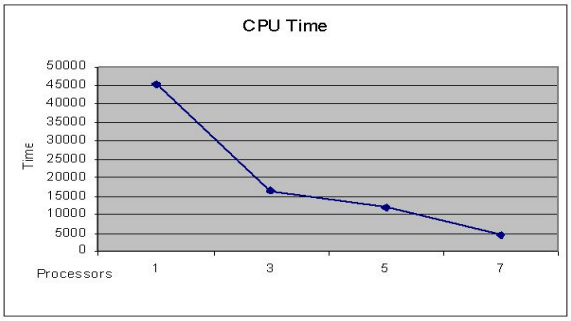


Figure 7. Parallel CPU Time

Table 1. Sequential versus Parallel Mehtods.

Processors	CPU Time (sec)	Speedup	Efficiency
1	45180		
3	16292.91	2.7730	0.9243
5	11743.47	3.8472	0.7694
7	4150.54	10.8853	1.5550

Table 2. Parallel TUNNEL-TGN behaviour.

5. Conclusions

The need to generate a set of optimal solutions has been shown, as the last minimum, with the lowest value of the objective function, may not give the best approximation. It is the set of solutions that gives the neccessary information for decision taking. The same is true for each individual well and for the field forecast.

The parallelisation of the method accelerated the procces 17 times, presenting the Parallel Tunneling Method as a promising toool for real History Matching applications.

Acknowledgements. We want to thank Schlumberger Mexico, for letting us use a multiple ECLIPSE license, on a cluster machine. Also, we thank the Supercomputo department at DGSCA UNAM for the use of their parallel machines to test the parallelisation scheme. We acknowledge the help we received from Francisco J. Cardenas to be able to run ECLIPSE on the cluster he supervises.

References

[1] Barron, C. and Gomez, S., *The Exponential Tunneling Method*, Reporte de Investigacin IIMAS, Vol. 1, No. 3, 1991.

[2] Bissel, R.C., *Calculating Optimal Parameters for History Matching*, Proc., 4th European Conference on Mathematics of Oil Recovery, Rros, Norway, 7-10 June 1994.

[3] Brun, B., Gosselin, O., and Barker, J.W., *Use of Prior Information in Gradient-based History Matching*, paper SPE 66353 presented at the 2001 SPE Reservoir Simulation Symposium, Houston, 11-14 February 2002.

[4] Castellanos, L. and Gomez, S., *A New Implementation of the Tunneling Methods for Bound Constrained Global Optimisation*, Reporte de Investigacin IIMAS, Vol. 10, No. 59. pp. 1-18, 2000.

[5] Castellanos, L., Gomez, S. and Guerra, V., *The Triangle Method to Locate the Corner of the L-Curve*, Journal of Applied Numerical Mathematics, Vol. 43, No. 4. pp. 359-373, 2002.

[6] Giliazov S.F. and Goldman N.I., *Regularisation of ill-posed problems by iteration methods*, Kluwer Academic Publishers, 2000.

[7] Gill, P., Murray,W. and Wright, M.H., *Practical Optimisation*, Academic Press, 1981.

[8] Gomez, S., Del Castillo, N., Castellanos, L. and Solano, J., *The Parallel Tunneling Method*, Parallel Computing Vol. 29, pp. 523-533, 2003.

[9] Gomez, S., Gosselin, O. and Barker, J., *Gradient-Based History-Matching with a Global Optimisation Method*, SPE Journal, pp. 200-208, June 2001.

[10] Gomez, S., Ono, M., Gamio, C. and Fraguela, A., Reconstruction of Capacitance Tomography Images of Simulated Two Phase Flow Regimes, J. of Applied Numerical Mathematics, Elsevier, Vol. 46(6), pp. 511-518, 2003.

[11] Gomez, S., Solorzano, J., Castellanos, L. and Quintana, M.I., *Tunnelling and Genetic Algorithms for Global Optimisation*, in . Hadjisavvas and Pardalos (eds.), Advances in Convex Analysis and Global Optimisation. Non Convex Optimisation and its Applications, Kluwer Academic Pub. pp. 553-567, 2001.

[12] Gomez, S., Perez, A., Dilla, F and Alvarez, R.M., *On the Automatic Calibration of a Confined Aquifer*, Computational Methods in Water Resources, XIV, Kluwer Academic Pub., 2002.

[13] Kelley C.T., *Iterative methods for linear and nonlinear equations*, Frontiers in Applied Mathematics, Vol. 16, SIAM, 1995.

[14] Levy, A. and Gomez, S., *The Tunneling Method Applied to Global Optimisation*, in P.T. Boggs, R.H. Byrd and R.B. Schnabel (eds.), Numerical Optimisation, SIAM, pp. 213-244, 1985.

[15] Levy, A. and Montalvo, A., *The Tunneling Algorithm for the Global Minimisation of Functions*, SIAM J. Sci. Stat. Comput., Vol. 6, No. 1, pp. 15-29, Jan. 1985.

[16] More, J.J. and Chih-Jen, L., *Newton's Method for Large-Scale Bound Constrained Optimisation Problems*, SIAM Journal on Opt. Vol. 9, No. 4, pp. 1100-1127, 1999.

[17] *The PUNQ Project*, URL: www.nitg.tno.nl/punq/

Parallelization of an algorithm for finding facility locations for an entering firm under delivered pricing

J.L. Redondo[a], I. García [a], P.M. Ortigosa [a], B. Pelegrín [b], P. Fernández[b]

[a]Dpt. Computer Architecture and Electronics, University of Almería, Spain.

[b] Dpt. Statistics and Operational Research, University of Murcia, Spain.

Abstract

This work presents the parallelization of an algorithm for finding facility locations for an entering firm which has to make decisions on the locations of its facilities as well as on its price setting in order to maximize profit. This combinatorial location problem is solved by *GASUB*, a new multimodal genetic algorithm with subpopulation support. The high computational requirements of the location problem demands the parallelization of the method. In this work two standard strategies have been implemented and compared. The first one follows a *master-slave* model, and the second strategy is a *coarse-grain* parallelization.

Keywords: Competitive location, Global optimization, Evolutionary computing and genetic algorithms, Parallel strategies

1. Introduction

Location decisions are frequently made by an entering firm that has to compete for customers with other pre-existing firms in the market. This has led to a variety of locations models with the aim of finding strategic locations for profit maximization (see [1]). These models are often extremely difficult to solve, at least optimally. Even the most basic models are computationally intractable for large problems instances, as it happens when the firm has to select a number s of locations in a set of m potential sites. In this case, the firm has to explore a large number ($\frac{m!}{s!(m-s)!}$ combinations) of possible locations to find an optimal solution. This is a combinatorial optimization problem and is hard to solve for high values of m and s.

We are interested in solving the problem of finding facilities location for an entering firm under delivered pricing. In this location model, the firms offer to sell the product to customers and also provide delivery for a single bundled price. Then customers buy from the facility that offers the lowest price in the area they belong to. In this setting, the entering firm have to make strategic decisions on location and price for maximizing profit. As a result of price competition, it is shown that a price equilibrium exists for any set of fixed facility locations (see [3]). Then the location-price decision problem is reduced to a new location problem (see [2]).

This work describes a new genetic algorithm with subpopulation support (GASUB), which inherits some features of UEGO ([4,5]), a stochastic optimization algorithm. GASUB is a general algorithm for solving combinatorial optimization problems which has proved its capabilities for finding optimal solutions for the location problem. In [6] GASUB is evaluated and compared to one of the standard software tools (Xpress-MP [7]) frequently used to deal with this kind of problems. It has been shown that GASUB can compete with Xpress-MP in finding global optima. For very hard problems Xpress-MP was not able to reach a solution while GASUB was. However, when the number of new locations increases, the execution time for GASUB becomes unacceptable, so parallel implementations can help to eliminate this drawback.

GASUB was designed with parallelism in mind, so its parallel implementations do not need complicated parallel models but the simple master-slave or the coarse grain strategies are good enough to obtain efficient solutions for problems of a very high computational cost. This work explores and evaluates two parallel implementations of GASUB with application to complex location problems.

In the following sections more details of this interdisciplinary problem will be provided. The problem of finding facilities location for an entering firm is described in Section 2. Section 3 contains a short description of GASUB, the genetic optimization algorithm. Details of our parallel implementations of GASUB and their evaluations on a cluster of processors are the topics of Sections 4 and 5, respectively. Finally, Section 6 draws some conclusions.

2. The entering firm location-price problem

We consider a set $C = \{1, 2, ..., n\}$ of spatially separated market areas within a region R. Customers in area i are aggregated at a market point v_i in R. Demand for a homogeneous product is assumed to be known and fixed, being w_i the amount of product required at v_i.

The product will be served by a set of facilities $F = \{1, 2, ..., \bar{q}\}$, the first q ($1 < q < \bar{q}$) already exist in the region and the others $s = \bar{q} - q$ are to be located. Each facility j is located at a point f_j in R. The points f_1 to f_q are known and the points f_{q+1} to f_m have to be selected in a set L of locations, which contains m potential sites. All the market and the location points are nodes in a transportation network immerse in R, where the product will be delivered. With d_{ij} we denote the distance between points v_i and f_j, the transportation cost per unit of product and unit of distance is denoted by t. Each facility j will deliver the product to customers in v_i at a unit price p_{ij} which include the transportation cost. It is assumed that all facilities have the same production cost p_{prod}, and that $p_{ij} \geq p_{min} + td_{ij}$, where p_{min} is the minimum price per unit of product (greater than the production cost).

Customers buy at the facility that offers the lowest price in the area they belong to. If two or more facilities set the lowest price at some market point v_i, customers in area i buy at the closest facility. Furthermore, since this assumption does not prevent that, when more than one of the cheapest facilities is also the closest, ties still may occur, we consider that half of the market in area i is captured by the new firm.

As a result of price competition (See [2] for an analysis on the process of price changing), the entering firm can only capture market at the points v_i such that $d_i^S \leq D_i$, where S denote the set of locations for the new facilities ($S \subset L$), $d_i^S = \min\{d_{ij} : f_j \in S\}$ and $D_i = \min\{d_{ij} : j = 1, ..., q\}$. The optimal price at v_i offered by the new firm is $p_i^S = p_{min} + tD_i$. Let $C_S^1 = \{i \in C : d_i^S < D_i\}$, $C_S^2 = \{i \in C : d_i^S = D_i\}$ and $p_{net} = p_{min} - p_{prod}$. Then, the maximum profit the new firm gets by locating its facilities in S is :

$$\Pi(S) = p_{net}\Big(\sum_{i \in C_S^1} w_i + \frac{1}{2}\sum_{i \in C_S^2} w_i\Big) + t\sum_{i \in C_S^1}(D_i - d_i^S)w_i$$

In order to define a search domain and the structure of the points defined in this domain, this combinatorial problem must be encoded. A point (individual in terms of genetic algorithms) consists of a single string that is a collection of m bits. The position of a bit in the string coincides with the index of the associated facility. Each bit can have 0 or 1 values, where 1 indicates that the associated facility has been chosen as part of a solution. As the set S of selected facilities is predetermined for every problem, the number of bits to 1 ratio must be fixed to the number of new facilities to be selected (cardinal of S). We must consider this constraint when generating any search point.

3. GASUB: A genetic algorithm for global optimization

GASUB is a genetic algorithm with subpopulation support, where every subpopulation is intended to occupy a local maximizer of the fitness function with no a priori knowledge of the total number of local optima in the domain function. This means that when the algorithm starts running it does not know how many subpopulations will appear, so it is necessary to have a mechanism to create and fuse subpopulations. Thus, new subpopulations are created when it is likely that the parents are on different hills, and subpopulations have to be fused when they are thought to climb the same hill.

To illustrate the way GASUB that copes with unevenly spread optima, it is natural to use a terminology that is well known from the field of simulated annealing. Thus, when illustrating our definitions and methods, we will talk about the 'temperature' of subpopulation, the ability of escaping from local optima. In our system, we made the 'temperature' an explicit attribute of every subpopulation (it is the attraction of subpopulations). This allowed us to offer an algorithm that 'cools down' the system while subpopulations of different 'temperatures' are allowed to exist at the same time. The basic idea of the algorithm is that 'cooler' subpopulations are allowed to create 'warmer' subpopulations by autonomously discovering their own local area of attraction.

A key notion in GASUB is that of a *subpopulation*. A particular subpopulation is not a fixed part of the search domain; it can move through the space as the search proceeds. A subpopulation would be equivalent to a single individual, which is defined by a center, a fitness function and a radius value. The center is a solution and the radius indicates the attraction area of this subpopulation (temperature). This definition assumes a *distance* defined over the search space. For our combinatorial problem we define the *Hamming distance*. Because of the constraint of the problem, the number of chosen facilities and hence the number of bits (or genes) to 1 is fixed, so the *Hamming distance* between any two feasible points (individuals) must be always multiple of 2.

The radius of a subpopulation is not arbitrary; it is taken from a list of decreasing radii, the *radius_list*. The radii decrease in a regular fashion in geometrical progression. The first element of this list is always the diameter of the search space (r_1), which will ensure that the largest subpopulation always contains the whole space independently of its center. The diameter is given by the largest distance between any two possible solutions according to the distance mentioned above, and it is an input parameter. If the radius of a subpopulation is the ith element of the list, then we say that the *level* of the subpopulation is i.

During the optimization process, a list of subpopulations is kept by GASUB and this *subp_list* defines the whole *population*. The maximal length of the *subp_list* is given by *max_subp_num*, which is an input parameter that indicates the maximum population size. The algorithm GASUB is in fact a method for managing the *subp_list* (i.e. generating, selecting and mutating subpopulations). See Algorithm 1.

At the *initialization* part a single subpopulation with a random individual as center is created. This individual must have the same ratio of genes to 1 as the number of new facilities and its associated radius will be equal to the diameter of the search space.

The *mutation* procedure applies consecutive interchanges of one facility to every center of every subpopulation. At the *generation* procedure, when it is known that more than one local maximum exists inside the subpopulation, each subpopulation in the list is divided in two or more.

The *selection* procedure has two mechanisms, the first one tries to fuse subpopulations that are too close, so if the centers of any pair of subpopulations from the *subp_list* are closer to each other than the radius associated to the current level, then these subpopulations are fused. The center of the new subpopulation will be the one with the best function value while the radius will be the largest one. The second mechanism selects the subpopulations to be maintained in the *subp_list*. If the

Algorithm 1 GASUB Genetic algorithm with subpopulation support

```
1 proc GASUB ≡
2   Initializing_population
3   Mutation(n_1)
4   for i := 1 to levels
5       Determine(r_i, new_i, n_i)
6       Generation_and_crossover(new_i/length(subp_list))
7       Selection(r_i, max_subp_num)
8       Mutation(n_i/max_subp_num)
9       Selection(r_i, max_subp_num)
```

number of subpopulations overcomes the allowed maximum number, the subpopulations that have been created more recently are eliminated.

GASUB is an iterative algorithm with *levels* iterations and with a budget determined by the maximum number of function evaluations (N). Both, *levels* and N are users-given parameters.

At each level, in addition to the radii (r_i), two important parameters are computed; the number of function evaluations for subpopulations generation (new_i) and the mutation (n_i). These principles are described in detail by [4,5].

4. Parallel strategies

The high computational requirements of the optimization algorithms have raised the appearance of numerous parallelization strategies. In this work two parallel strategies have been implemented, an asynchronous *master-slave* model and a *coarse-grain* model.

4.1. Asynchronous master-slave strategy

In the *master-slave* model (PAGASUB) the parallelism comes from the evaluation of the individuals in the population. This is due to the fitness of an individual being independent from the rest of the population, and there is no need to communicate. The evaluation of individuals is parallelized by assigning a fraction of the population to each available processor. Communication occurs only as each slave receives its subset of individuals for evaluation and when the slaves return the fitness values. If the algorithm stops and waits to receive the fitness values for the entire population before proceeding into the next generation, then the algorithm is synchronous. We have implemented an asynchronous *master-slave* algorithm where the algorithm does not stop to wait for any (slow) processors.

The *slaves* processors only need to receive the two features of any subpopulation (i.e. its center and its radius), from the *master* processor. In order to be able to run the *mutate* and *generate* procedures, so the amount of information involved in the communication procedures is quite small. Both procedures do not need any additional information; they depend only on a single subpopulation. For this reason, these procedures can be run independently.

At the initialization phase, the *master* processor creates the initial *subp_list* containing several individuals. Then it distributes the species among the *slave* processors so all subpopulations can be optimized simultaneously. When the slaves finish, they send the results to the *master* processor and it updates the list of subpopulations that will be used in the following level. Then an iterative process (*levels*) is carried out (See Figure 1).

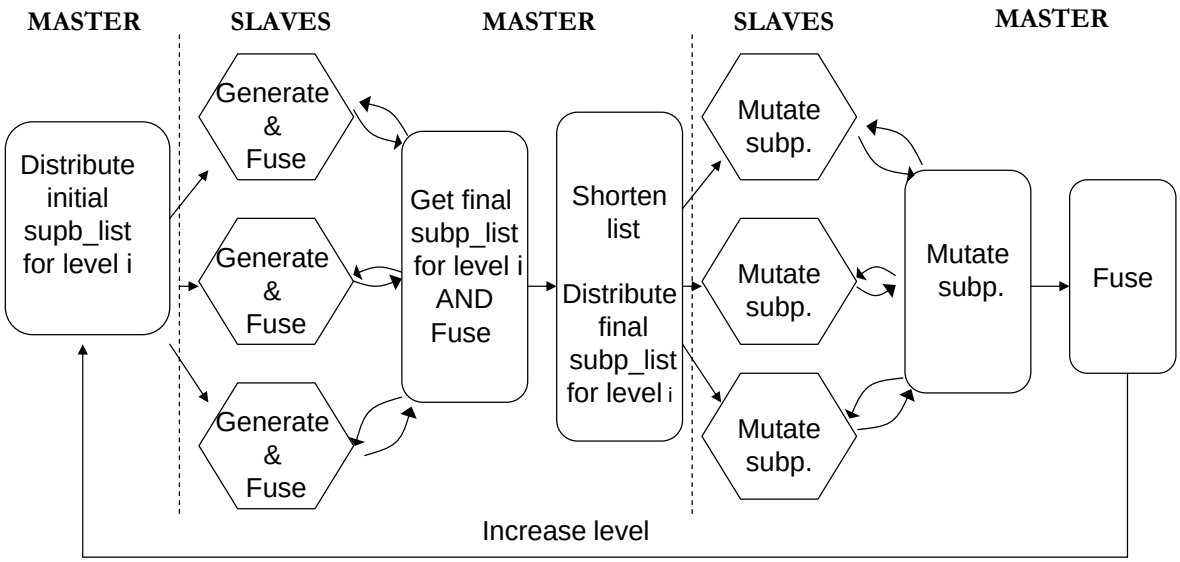


Figure 1. Asynchronous master-slave model

Rectangular and hexagonal boxes represent commands executed by the *master* and the *slaves* processors, respectively. Vertical dotted lines indicate synchronisation points and arrows represent communications.

Initially, the *master* processor distributes the *subp_list* among the *slave* processors. The *slave* processors pick up the subpopulations and evaluate them, trying to create new subpopulations. When the *slaves* finish the generation procedure, they fuse their list of subpopulations before sending them to the *master* processor.

In a synchronous version, the *master* processor stays in a wait state until **all** the *slave* processors finish generating subpopulations. Once the *master* processor has received all the new subpopulations, it applies the fusion and selection processes to them in order to complete the final *subp_list* at this level. Later, it distributes this list among the *slave* processors, which take charge of mutating each of their assigned species. When all species have been mutated, they are sent to the *master* processor that applies a fusion process and forms the *subp_list* that will be used in the following iteration.

In this case, the distribution of the computational load is not well balanced. The evaluation of the objective function is carried out within the subpopulation generation and mutation processes, which are only executed by the *slave* processors. Therefore, the *master* processor is mostly waiting for results from the *slave* processors. Due to the fact that the performance of this synchronous version is very low, elimination of some of synchronisation points is necessary. In this way the *master* processor can also work on the generation and mutation processes and the processors can distribute the computational load in a more dynamic way.

In the asynchronous implementation (PAGASUB) solves some of the previous problems. In this implementation the load has been balanced forcing the *master* processor to mutate and generate subpopulations while the *slave* processors are working. Now the *master* processor can start to carry out the synchronous operations over the species before it has received all the information, so the idle time can be reduced considerably.

Algorithm 2 CGGASUB Coarse-grain model

```
1  proc CGGASUB ≡
2    Initializing_population
3    Mutation(n_1)
4    for i := 1 to levels
5      Determine(r_i, new_i, n_i)
6      Generation_and_crossover(new_i/length(subp_list))
7      Selection(r_i, max_subp_num)
8      Mutation(n_i/max_subp_num)
9      Selection(r_i, max_subp_num)
10     if (i > levels/2)
11       Migrate_subpopulation
```

The *master* processor is constantly checking for the arrival of information (a new generated sublist of subpopulations or a mutated subpopulation) from the *slave* processors and if any, it immediately sends back a new species. Otherwise the *master* processor contributes to the optimization process by fusing or mutating the received subpopulations. These processes executed by the *master* processor are always interrupted when new information arrives from any *slave* processor. When all subpopulations have been mutated, the master processor applies a fusion process and creates the subpopulation list that will be used in the following iteration.

4.2. Coarse-grain strategy

In the *coarse-grain* model (CGGASUB), each processor executes the GASUB algorithm on different subpopulations *subp_list* in an independent way. Nevertheless, intermediate results (subpopulations) are sometimes exchanged among processors. The *coarse-grain* structure is shown in the Algorithm 2.

In the *Migrate_subpopulation* process, each processor sends all other processors the subpopulation with the best fitness or function value. This migration process is only carried out at the half last levels. Migration is not carried out at early phases of the algorithm in order to allow the populations to evolve independently. Taking into account that establishing all to all communications (broadcast) is very expensive when the number of processors raises, the migration has been implemented using a recollector processor, which receives the individuals from the remaining processors, joins them into a sublist and sends this to all processors.

In this parallel implementation, only one of the input parameters is modified regarding to the sequential version: For each computer, the size of *subp_list* is obtained by dividing the initial list size between the number of processors.

5. Experiments

The parallel algorithm has been implemented using C++ and the message passing library MPI. The experiments have been carried out on a cluster, which consists of 52 900 MHz Ultrasparc III processors in a single cabinet with 1 Gb of memory per processor, totally 52 GB. The level 1 cache on the UltraSPARC-III is 64 KB, 4-way set-associative with 32 byte lines. The level 2 cache on the UltraSPARC-III is 8 MB, direct-mapped with 64 byte lines.

For computational experiments we have selected 1046 cities as nodes, and their populations as

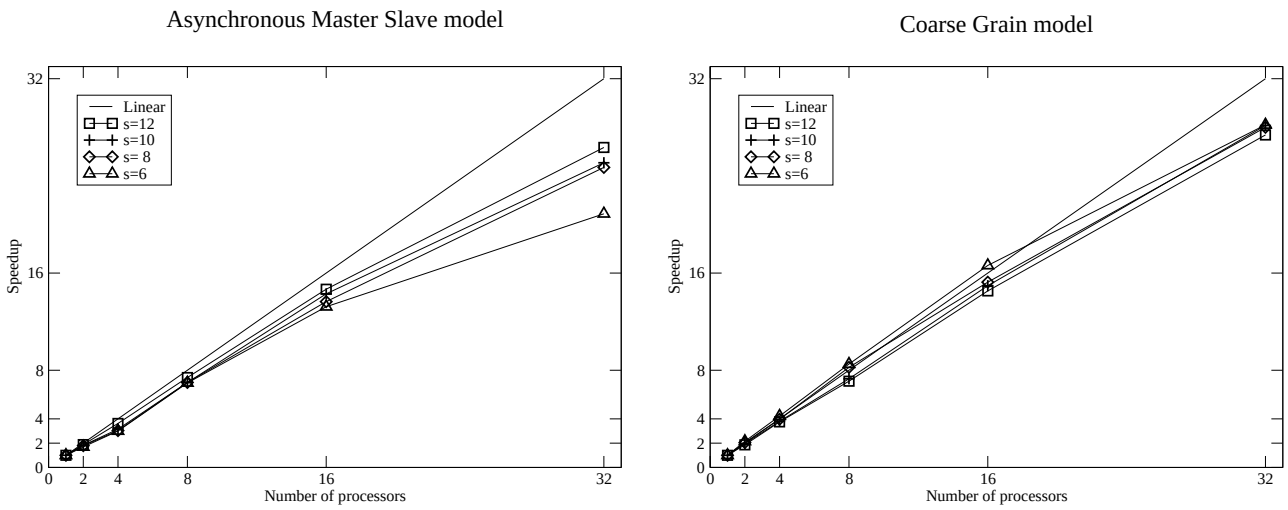


Figure 2. Speed up Asynchronous master-slave model (left) Coarse-grain model (right)

demand. The number of pre-existing facilities was equal to $q = 10$, the number of new facilities were equal to $s = 6, 8, 10, 12$, the unit transportation cost was equal to $t = 0.001$. Finally, we fixed $p_{net} = 1$. More details about the features of these experiments are shown in [6], where results from the sequential version are compared to results obtained when solving the problem with Xpress-MP, an integer linear programming optimizer. In this section these experiments have been executed and evaluated by the parallel versions. The experiments were executed using P=2^i processors, with $i = 0, 1, 2, 3, 4, 5$.

Due to the stochastic nature of the algorithms, all results given in this work are average values of ten executions, obtaining a statistic ensemble of experiments. In order to measure the reliability and efficiency of the parallel algorithms, the obtained results have been compared to the sequential ones. First, the convergence to optimal solutions with the same precision than sequential version was verified, so the quality of the algorithms remains in the parallel versions. The number of evaluations in the asynchronous master-slave parallel version are similar to the sequential ones. However, in the coarse-grain technique the number of evaluations decreases lightly, due to the user given parameters approximation.

The execution time when varying the number of processors has been also computed, and speedup analysis carried out. Figure 2 shows the efficiency of the asynchronous master-slave and coarse-grain models.

In the *master-slave* model the number of communications depends on the complexity of the problem and stage of the algorithm, while in the *coarse-grain* parallel algorithm is fixed and proportional to the number of processors and the *levels* parameter. In this way, although both strategies do the same amount of work (number of evaluations), the communication cost increases the total execution time in the master-slave model. Consequently the *coarse-grain* model obtains better values for the speedup than the *master-slave* model.

In the coarse grained decompositions there are some cases where super linear speedup are obtained. This is due to number of function evaluations decreasing. With respect to the communication costs, in this model increasing the complexity of the problem (from s=6 to s=12) causes more uneven loading of the processors, and thus the communication takes longer (See Figure 2 (right)).

Nevertheless, in the *master-slave* model, even though the amount of work each process does

increases (from s=6 to s=12), the distribution of the load is well balanced and the communication costs do not increase as a result of complexity increment of the problem. So, the communication time is compensated by the computational time, improving the obtained speedup, as we can see in Figure 2 (left).

6. Conclusion

In this work we have proposed a new genetic-like algorithm, for finding solutions to different facility locations problems which are hard to solve. Taking into account the intrinsic parallelism of the algorithm, two standard strategies have been implemented and evaluated. Both strategies obtain the same quality of results and their corresponding percentage of success on finding the global solution is 100%.

The asynchronous master-slave model obtains a good speedup thanks to the dynamic distribution of the load and the participation of the master processor in the tasks with more computational complexity. The course-grain model obtains good speedup thanks to the intrinsic parallelism of the sequential algorithm. However, a light load imbalance occurs causing some uneveness in communication costs.

7. Acknowledgments

This research has been supported by the Ministry of Science and Technology of Spain under the research projects BEC2002-01026 and CICYT-TIC2002-00228, in part financed by the European Regional Development Fund (ERDF). This work was carried out under the HPC-EUROPA project (RII3-CT-2003-506079), with the support of the European Community - Research Infrastructure Action under the FP6 "Structuring the European Research Area" Program).

References

[1] H.A. Eiselt, G. Laporte, and J.F. Thisse. Competitive location models: a framework and bibliography. *Transportation Science*, 27:44–54, 1993.

[2] P. Fernández, B. Pelegrín, M.D. García, and P. Peeters. A discrete long-term location-price problem under the assumption of discriminatory pricing: Formulations and parametric analysis. *European Journal of Operations Research*, pages –, 2005. forthcoming.

[3] M.D. García, P. Fernández, and B. Pelegrín. On price competition in location-price models with spatially separated markets. *TOP*, 12(2):351–374, 2004.

[4] M. Jelásity, P.M. Ortigosa, and I. García. UEGO, an abstract clustering technique for multimodal global optimization. *Journal of Heuristics*, 7(3):215–233, 2001.

[5] P.M. Ortigosa, I. García, and M. Jelasity. Reliability and performance of UEGO, a clustering-based global optimizer. *Journal of Global Optimization*, 19(3):265–289, 2001.

[6] B. Pelegrín, P. Fernández, J. L. Redondo, I. García, and P. M. Ortigosa. Finding locations for an entering firm under delivered pricing competition. http://www.ace.ual.es/Investigacion/papers/05/report11.pdf. Technical Report TR-11, Departamento de Arquitectura de Computadores y Electrónica, Universidad de Almería, 2005.

[7] Xpress-MP. *Dashoptimization*, 2004.

Concurrent parallel shortest path computation *

Doron Nussbaum, Jörg-Rüdiger Sack and Hua Ye [a]

[a]School of Computer Science, Carleton University, Ottawa, Ontario K1S-5B6, Canada
e-mail: {nussbaum, sack, hye}@scs.carleton.ca

In this paper we introduce the notion of concurrent parallelism for the fundamental problem of answering shortest paths queries between pairs of points located on a weighted polyhedron or terrain. We discuss design and implementation of our algorithm and show experimentally, that the efficiency and speed-up for parallel shortest paths queries can be increased through concurrency. Our work also represents the first implementation of the bisector algorithm which currently has the best theoretical time complexity for computing weighted shortest paths.

1. Introduction

The computation of shortest paths arises in application areas such as geographical information system (GIS), network routing, and robotics. It ranks among the fundamental problems studied in computer science, graph theory and computational geometry. Our motivation came from search & rescue, where a rescue team needs to reach the location of an accident as fast as possible, i.e., using the shortest path. Typically in such problems, the domain is a terrain represented as a triangular irregular network, called TIN. The cost of travel in terrains may differ from triangle to triangle. For example, one triangle may represent water while another may represent a forest. To capture travel through such terrains, the Euclidean distance is not an appropriate distance measure. The weighted shortest path problem arises when we associate a positive weight with each triangle. Unlike in the Euclidean setting, weighted shortest path computations on terrains are both time consuming and complex to solve. No exact algorithm exists and even the first approximation algorithm [13] discovered was of limited practical value (even if it had been implemented) as the time complexity was approximately $O(n^8)$ for a TIN consisting of n weighted faces. In order to speed up the computation we follow a two-prong approach (1) developing more efficient approximation techniques (the reader is referred to the most recent work, e.g., [5], and for a comparison of existing sequential algorithms) and (2) designing parallel algorithms.

For Euclidean shortest paths computations, a variety of parallel algorithms can be found in the literature [1,6,8–10,12,16]. For weighted shortest paths computations, given the high sequential time complexities and its practical relevance, parallelization is appropriate or even required which motivates the work described here and our earlier work discussed in [12]. The differences between existing parallel algorithms are: (a) types of parallel architectures, (b) particular problem variants (e.g., one-to-one and one-to-all), (c) objects/domains, and (d) implementation. Parallel one-to-one shortest paths algorithms are usually based on a variation of Dijkstra's shortest path algorithm [7]. Such parallel algorithms, which are designed for distributed memory computers, partition the data among the processors and then execute Dijkstra's algorithm locally on each processors. When the shortest path processing reaches the boundary of a partition, processors send messages to neighbouring processors to update their local information (see [12] for more details of a parallel implementation).

The shortest path problem appears to be sequential in nature and the performance of a parallel implementation degrades as the number of processors increases [9,12]. In an information system accessed by many users, multiple weighted shortest path queries must be processed (e.g., for planning several approaches in search&rescue or in a tourist information system). Using a parallel computer these queries would be processed sequentially and the response time of each query depends on the order in which the queries arrive. Often when a shortest path query is processed on a medium to large number of processors (16-128 processors) the idle time of each processor increases. This leads to a reduction in utilization and efficiency of the processors as was observed by [9,12], where the efficiency for p processors was about $\sqrt{p}$.

This paper discusses an implementation and experimental study of concurrent parallelism for shortest path queries on weighted polyhedra or TINs. Our objective is to demonstrate an increase in processor utilization and speed-up. The paper is organized as follows: in Section 2 we sketch the bisector ϵ-approximation technique of [4,5] used in this implementation. In Section 3 we discuss concurrent parallelism. In Section 4 we show experimental results and we conclude with Section 5.

*Research supported in part by NSERC and Sun Microsystems of Canada

2. Formal Problem Definition and Background

We first define the problem formally. Let $\mathcal{P} = \{f_1, f_2, \ldots, f_n\}$ be a continuous 2.5D polyhedral surface (*terrain*) composed of n triangular faces $f_i, 1 \leq i \leq n$. Each face $f_i, 1 \leq i \leq n$, has an associated positive weight w_i which determines the cost of travel through that particular face. Given such a terrain $\mathcal{P}$, and two points s, t on $\mathcal{P}$, a weighted shortest path $\Pi(s,t)$ between s and t is defined to be a path of minimum cost among all possible paths joining s and t that lie on the surface of $\mathcal{P}$. The cost of the path $c(\Pi(s,t))$ is the sum of the Euclidean lengths of all path segments multiplied by the corresponding face weight $c(\Pi(s,t)) = \sum_{i=1}^{k}(|s_i| \times w(f(s_i)))$, where k is the number of segments of the path; $|s_i|$ is the Euclidean length of the i^{th} segment s_i; $f(s_i)$ is the face that contains segment s_i and $w(f(s_i)))$ is the weight of face $f(s_i)$. (When all weights are equal the Euclidean shortest path problem is obtained.)

The determination of weighted shortest paths on polyhedral surfaces (or in 3-space) has been studied extensively by researchers over the past approximately 20 years; only approximation solutions are available and some problem instances are known to be $\mathcal{NP}$-complete. Most of the currently best approximation algorithms utilize a discretization approach. They convert the continuous problem into a discrete problem creating an approximation graph $G^* = (V^*, E^*)$; and then execute Dijkstra's shortest path algorithm [7] between two points in G^* corresponding to the two inputs points on the surface. The approach must guarantee that the *approximating path* so constructed provides an approximation to the true shortest path. The approximation path is called ϵ-*approximation*, $\epsilon \geq 0$, of the true shortest path, if its cost is at most $(1+\epsilon)$ times the cost of the real shortest path.

We briefly sketch the discretization of $\mathcal{P}$: first Steiner points are strategically added to $\mathcal{P}$ which, together with the vertices of $\mathcal{P}$, form the vertex set V^*. Once V^* has been created, the edge set E^* is defined by interconnecting vertices of V^*. The choice of interconnection pattern and its size are crucial. Typically edges connect (selected) pairs of vertices located within one face or on at most two adjacent faces. The cost associated with each such edge then corresponds to the cost of the weighted shortest path between the two points restricted to lie inside a face (or inside two adjacent faces).

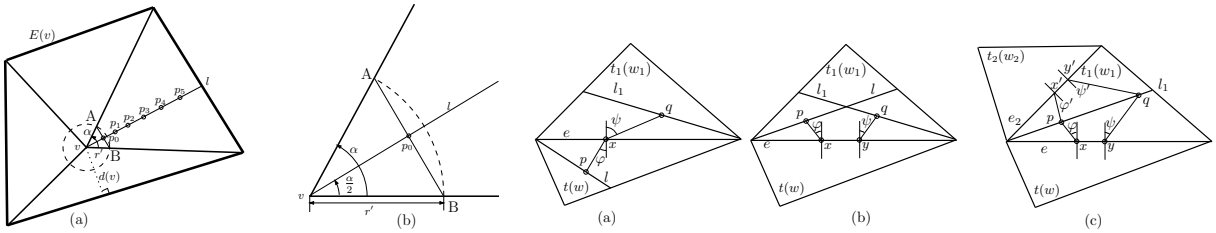
The performance of an approximation algorithm in terms of execution time and path cost accuracy is strongly correlated to the number of Steiner points in V^* and the edge interconnect scheme. In general, when the number of Steiner points in G^* increases, the accuracy of the approximated solution improves but the execution time increases. The challenging task is to develop very efficient discretization techniques while maintaining a high accuracy of approximation.

For weighted shortest paths on a non-convex polyhedral surface, Mitchell et al. [13] first presented an $O(n^8 \log \frac{n}{\epsilon})$ time ϵ-approximation algorithm using what they call "continuous Dijkstra" paradigm. Lanthier et al. [11] and Aleksandrov et al. [2,3] adopted a natural approach to discretize the surface by placing Steiner points on the TIN edges (see Section 2.1 for details). Lanthier et al. [11] produce an approximate shortest path $\Pi'(s,t)$, which holds $||\Pi'(s,t)|| \leq \beta(||\Pi(s,t)|| + W|L|),\ \beta > 1$ where W is the maximum weight among all face weights of the surface and $|L|$ is the Euclidean length of the longest edge of the surface. It runs in $O(n^3 \log n)$ time in the worst case, but performs very well in practice (see [11] and [17]). The ϵ-approximation algorithm presented by Aleksandrov et al. [3] runs in $O(\frac{n}{\epsilon} \log \frac{1}{\epsilon}(\frac{1}{\sqrt{\epsilon}} + \log n))$ time. Reif and Sun [15] used their approach and introduced a nice variation of Dijkstra's algorithm to obtain an improved $O(\frac{n}{\epsilon} \log \frac{n}{\epsilon} \log \frac{1}{\epsilon})$ bound. Recently, Aleksandrov et [4,5] presented a further improved ϵ-approximation algorithm that runs in $O(\frac{n}{\sqrt{\epsilon}} \log \frac{n}{\epsilon} \log \frac{1}{\epsilon})$ time. (Note that the improvement by a multiplicative factor of $\sqrt{\epsilon}$ is very significant.) The latter algorithm is called bisector ϵ-approximation algorithm in this paper, because it places the Steiner points on the angular bisectors of the triangle faces. The detail of this algorithm will be discussed in the next section. The algorithm [5] improves over the first approach [13] by approximately a factor of n^7, but has a higher dependency on the geometry of the input. The reader is referred to [5] for a full analysis of the algorithm including a detailed discussion of geometric parameters influencing the run-time.

2.1. Bisector Approximation

Our implementation is based on the most recent algorithm, the bisector ϵ-approximation method by Aleksandrov et al. [4,5] sketched next. We discuss the placement of the Steiner points on $\mathcal{P}$ and the edge connectivity used to form G^*. In this scheme the Steiner points are placed on the angular bisectors of each triangle and the edges of E^* are connecting these Steiner points (crossing face boundaries). We define these edges and their costs between any pair of Steiner points (or original vertices) p and q lying on neighbouring bisectors (two bisectors are neighbours if the angles they split share a common edge of $\mathcal{P}$).

For each vertex v, we denote by $E(v)$ the set of edges in the triangles incident to v minus the edges incident to v (the dark outer polygon in Figure 1(1.a)) and by $d(v)$ the minimum Euclidean distance from v to the edges in $E(v)$. The first Steiner point on bisector l of α, is placed at the intersection of l and $\overline{AB}$ where A and B are the intersection of a circle of radius $r' = \epsilon \times r(v) = \epsilon \times \frac{d(v)}{7}$ and the edge of α (Figure 1(1.b)). The remaining Steiner points on l are placed at $|vp_i| = \lambda^i \times |vp_0| \quad for\ i = 1, ..., k$, where $\lambda = (1 + \sqrt{\frac{\epsilon}{2}} \sin \frac{\alpha}{2})$ (Figure 1(1.b)). Similarly we place the Steiner points on all other bisectors.



1 (a-b) Steiner point placement

2 (a-c) Paths used to define edge costs

Figure 1. Defining graph G^*.

There are three cases to consider when connecting a pair of nodes (p, q):

p **and** q **are on different triangles** (Figure 1(2.a)) - A pair of neighbouring bisectors l and l_1 lying on different triangles. The shortest path between p and q bends at a point x on e, so that the angles φ and ψ satisfy the Snell's law: $w \sin\varphi = w_1 \sin\psi$. The cost of the edge ($p$, q) in G^* is equal to $w|px| + w_1|xq|$.

p **and** q **are on different bisectors of the same triangle** (Figure 1(2.b)) - We only add an edge between p and q when $w < w_1$ and there exists two bend points x and y on e, which must satisfy $\varphi = \psi$ and $\sin\varphi = \sin\psi = w/w_1$. The cost of the edge ($p$, q) in G^* is $||pq|| = ||px|| + ||xy|| + ||xq|| = w_1|px| + w|xy| + w_1|yq| = w_1(|px| + |yq|) + w|xy|$.

p **and** q **are on the same bisector** (Figure 1(2.c)) - The situation is similar to the previous one, except that there are two candidates for the shortest path between p and q, one crossing e and the other crossing e_2. We take the shorter resulting distance as weight into G^*.

The discretization step converts the polyhedral surface into a graph G^* upon which one can execute any discrete shortest path algorithm (e.g., Dijkstra's [7]). However, the size of the resulting graph G^* may be quite large which has a significant impact on the running time. In the case of the bisector $\epsilon-$ approximation scheme the number of Steiner points, k, also depends on the geometry of the network. It has been shown that $k = \lfloor \log_\lambda \frac{|l|}{|vp_0|} \rfloor$, where $\lambda = (1 + \sqrt{\frac{\epsilon}{2}} \sin\frac{\alpha}{2})$, and α is the angle that the bisector splits [4] and thus k increases as α gets smaller.

In the next section we present a way to overcome the impact of this increase on the running time of the algorithm. (We also investigated alternate methods to reduce the number of Steiner points at only a small loss in accuracy.)

3. Concurrent Parallelism

Shortest path problems appear to fall into this class of problems are sequential in nature. Using Dijkstra's algorithm, a shortest path $\pi(s, u)$, from s to u, can only be computed after completion of the computation of shortest paths $\pi(s, v)$, where v is any vertex that satisfies $||\pi(s, v)|| < ||\pi(s, u)||$. This manifests itself for parallel shortest paths computations as idle times for a large number of processors. By choosing a good partitioning scheme we can get processors involved more quickly and thus reduce idle times of processors to a certain degree. However, the idle times are still significant, especially when the number of processors increases (the efficiency is roughly $\frac{1}{\sqrt{p}}$ as experimentally determined in [12]).

Existing parallel shortest path implementations deal with concurrent shortest path queries in batch mode, i.e., one query at a time. This implies that queries are waiting even though some processors might be idle. Our proposed concurrent parallelism aims to use these idle cycles to reduce the response time for queries (difference between query's submission time and its output time). The main idea is to overload the processors with queries in such a way that when a query q_i is processed and a processor is idle, it can start to process another query q_j until it is required to resume its work on query q_i. (Note that it requires more resources (e.g., memory).) Allowing the processors to hold and process multiple queries concurrently is called here *concurrent parallelism*. Therefore, our objective is to show that using concurrent parallelism one can process concurrent queries simultaneously to obtain higher speedup and efficiency.

Concurrent parallelism must obey two principles: (1) queries are processed in order of submission, and, (2) each individual query is itself processed as fast as possible in parallel. In our solution to the parallel shortest path problem we adhere to these two principles as follows: (a) each new query is tagged with a query id, and (b) the implementation is based on a parallel shortest path algorithm similar to that of [12].

The parallel shortest path algorithm we implemented, is designed for a distributed memory architecture (e.g., a Beowulf computer). It consists of a preprocessing phase and a computation phase. During the preprocessing phase the graph G^* is generated, partitioned into pages and distributed to the processors using the Multi-dimensional Fixed Partition (MFP) scheme developed in [14]. The computational architecture is based on a manager workers scheme where one processor is designated as a manager and the remaining processors are designated as workers. Although a Beowulf architecture does not have topology, the MFP mapping scheme enables a processor to compute for a given page q the id of the processors which "own" the pages adjacent to q (thus imposing an implicit topology on the Beowulf architecture). During execution time, the manager receives new queries and submits queries to workers. Once a query has been

completed the manager obtains the results and presents the final result to the user.

Each query is processed in parallel by all workers. Our parallel algorithm used to process a single query is build upon a variation of Dijkstra's shortest path algorithm. Each processor executes this algorithm using a priority queue which we denote by SPQueue for each query. This SPQueue uses the travel cost as its key. In our implementation the SPQueue is implemented as a min-heap where the top element of the heap represents the vertex to be processed next in that query.

To meet the first principle of concurrent parallelism, queries must be processed in order of arrival. Thus, if a processor contains data of two different queries, say q_i and q_j, it must process q_i first if q_i was submitted prior to q_j. To process multiple queries simultaneously, each query is assigned a query id in the order of arrival. The query id is used to prioritize the processing order in case a processor contains multiple queries.

In our design, we use one SPQueue for each concurrent query and a priority queue (denoted by QueryHeap) as a container for these SPQueues. At the beginning of each iteration of Dijkstra's algorithm, we only access the SPQueue with highest priority among all SPQueues currently stored in QueryHeap. The priority of a SPQueue in the QueryHeap is determined by a pair (query status, query id). When SPQueue of a query is empty, its status is IDLE, otherwise its status is WORKING. A SPQueue with WORKING status has higher priority in QueryHeap than a SPQueue with IDLE status. Among all SPQueues with the same status, the SPQueue with smaller query id has higher priority. During execution a processor receives messages including (1) update message which contains the new travel cost of vertices, (2) query termination message. An update message is processed using Algorithm 1. When a query termination message is received the processor removes the query from QueryQueue.

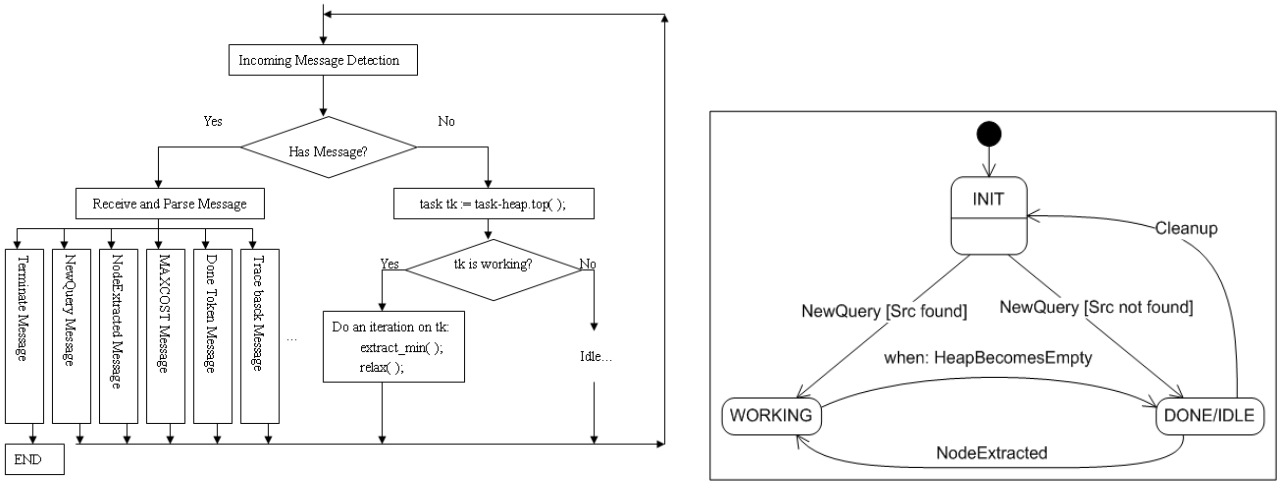
Algorithm 1 ProcessUpdateMessage()

```
1: if Query is not in the QueryQueue then
2:     Create a new SPQueue for the query
3:     Insert updated vertices to the SPQueue
4:     Insert the new query into the QueryQueue
5: else
6:     Locate the query in the QueryQueue
7:     Update the cost of the vertices
8:     Update the SPQueue of the query
9: end if
```

We observe from our experiments (see Section 4) that a relatively small number of concurrent queries are already sufficient to achieve a very good performance for our concurrent shortest path application. A skeleton of the execution of a worker processor is given in Algorithm 2 and depicted in Figure 2. Next we present the testing and the experimental results of concurrent parallelism.

Algorithm 2 ConcurrentSP()

```
1: for as long as there is work to do do
2:     if there is a message then
3:         if update message then
4:             Execute Algorithm 1
5:         else
6:             Remove query from the QueryQueue
7:         end if
8:     end if
9:     Obtain top element of QueryQueue and obtain its SPQueue
10:    Locally run Dijkstra shortest path algorithm of SPQueue
11:    Communicate with other processors to update cost and path information at partition boundaries
12:    if shortest path is found then
13:        Report the length
14:        Remove the query from the QueryQueue
15:    end if
16: end for
```



(1) Control Flow (2) State chart

Figure 2. Worker processor control flow and state chart.

4. Experimental Results

The objective of concurrent parallelism is to take advantage of idle cycles when executing a parallel application. We can only provide an overview of the extensive set of experiments conducted and present the results for random distribution of queries. The reader is referred to the full paper for a detailed discussion of the other (somewhat similar) results.

4.1. Test Setup

We tested the concept of concurrent parallelism using a parallel shortest path application. In our tests we examined a number of factors: parallel machine architecture, number of processors, type of queries, and concurrency level. We conducted our tests on a variety of spatial data and queries.

Machine Architecture - We employed a Symmetric Multi Processing (SMP) architecture - SunFire 6800 Cluster, and a distributed memory architecture - Pentium Xeon-based Beowulf Cluster. The SunFire had 20GB of memory, a 1.3 TB disk storage - Sun StorEdge[tm] T3 Disk array, and twenty 900 MHz UltraSPARC III Cu processors. The communication software was Sun Grid Engine v5.3 Enterprise Edition with Sun MPI v5.0. The Beowulf computer consists of 128 processors: (a) 32 nodes with dual 1.7 GHz Xeon processors, 1 GB RAM per node and 60GB disk (b) 32 nodes with dual 2.0 GHz Xeon processors 1.5 GB RAM per node and 60GB disk. The interconnect network is a Cisco 6509 switch with 1Gb cards for the nodes (the 1.7 GHz nodes use Intel Pro 1000 XT NICs and the 2.0 GHz nodes use on-board GigE interfaces). The OS was Redhat 7.3 using Sun Grid Engine Enterprise Edition with LAM-MPI v6.5.9.

- **The number of processors -** is 1, 4, 9, 16 corresponding to a mesh (torus) of size 1x1, 2x2, 3x3, and 4x4, respectively.
- **Type of queries-** queries are drawn from three query distributions: random distribution, and two cases of clustered distribution (see Figure 3).
  - **Clustered distribution -** tries to mimic spatial queries that relate to particular locations (e.g., downtown). We first randomly chose five vertices to represent five "city centres". Then, for each "city centre", we created a mixture of short, medium and long distance queries using two distance measures: link distance (clustered case 1) and face distance (clustered case 2). Using the link distance measure we conducted a one-to-all shortest path from each "city centre", say vertex s, to all other vertices. Let $d_v(s,u)$ denote the number of vertices along the shortest path from s to u and let w be the vertex such that $d_v(s,w) = \max d_v(s,u), u \in V^*$. A query from vertex s to u is a *short* query if $d_v(s,u) \leq \frac{d_v(s,w)}{3}$. Similarly, a query from vertex s to u is a *medium* (*long*) query if $\frac{d_v(s,w)}{3} < d_v(s,u) < \frac{2d_v(s,w)}{3}$ (if $\frac{2d_v(s,w)}{3} \leq d_v(v,u) \leq d_v(s,w)$). In the case of face distance we replaced $d_v(s,u)$ by $d_f(s,u)$ where $d_f(s,u)$ denotes that number of faces (triangles) intersected by the shortest path from s to u.

    We choose three test sets for each of the distance measures: (a) **Long-paths-dominated test (denoted by 5127) -** each source point is associated with one short query, two median queries and seven long queries;

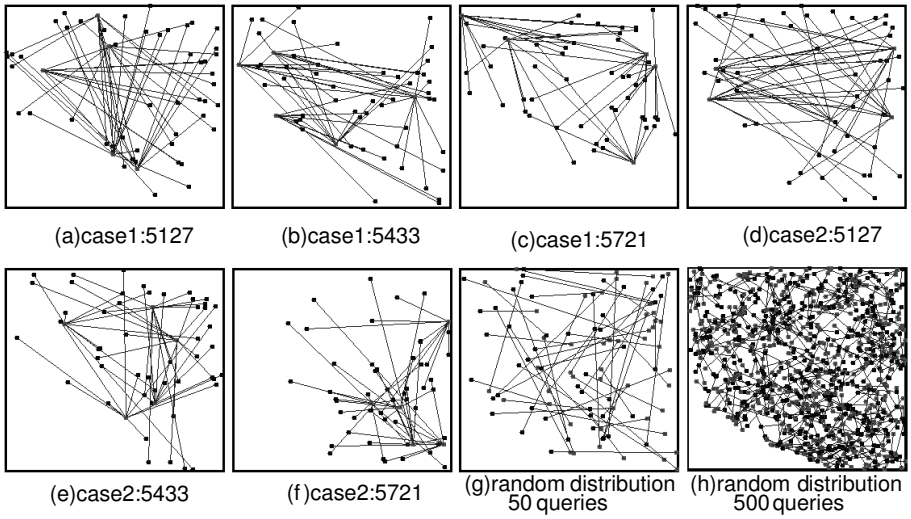


Figure 3. Query distributions - (a), (d) long-paths-dominated test sets, (b), (e) Balanced-distance test sets, (c), (f) short-paths-dominated test sets, and (g), (h) random distribution tests set with 50 queries and 500 short queries, respectively.

(b) **Balanced-distance test set (denoted by 5433) -** each source point is associated with four short queries, three median queries and three long queries; and (c) **Short-paths-dominated set (denoted by 5721) -** each source point is associated with seven short queries, two median queries and one long query. Therefore, each clustered distance measure consists of 50 queries.

– **Random distribution -** Two sets of sets of random queries were used: (a) 50 randomly generated query pairs, and (b) 500 randomly generated short distance queries.

- **Concurrency Level -** different concurrency levels where used 1, 5, 20 and 50.

We used a terrain with 5,000 vertices, 9,799 faces, and 14,798 edges using $\epsilon = 0.1$. The number of edges and vertices grows significantly due to the bisector ϵ-approximation. We chose this terrain to ensure that the data sets and the execution of the algorithm can fit into the memory of a single processor (eliminating any bias due to external memory use). To measure the performance of concurrent parallelism we use the traditional methods of speed up, $S = \frac{T_1}{T_p}$, and efficiency, $E = \frac{S}{p}$, where T_1 is the execution time on a single processor, T_p is the execution time on p processors and p is the number of processors. In addition we introduce a measure called response time to capture how long does it takes to obtain an output to a submitted query. Let T_a and T_f denote the time that a query is submitted and the time that an output is produced, respectively. The response time for the i^{th} query is defined as $T_{r_i} = T_{f_i} - T_{a_i}$, for $i = 1, \ldots, m$, where m is the total number of queries. The average response time $T_{rav} = \frac{1}{m} \sum_{i=1}^{m} T_{r_i}$ is a good indicator of the system performance.

4.2. Results Analysis

Tables 1 and 2 show the results of executing 50 random queries and 500 random short queries, respectively (see figures 3(g) and (h)). We draw attention to: speedup and efficiency, idle time, and response time. Recall that we are interested in the effect of submitting a number of queries simultaneously to the parallel application. Because of space limitations we depict only the results of Table 2 in Figure 4.

I. Speedup and efficiency (Figure 4.c) - here concurrent parallelism significantly improved the performance of the parallel shortest path application. In the case of 500 random short queries, we obtained about 100% improvement over sequential query submission. For example, sequential submission obtained a speed up of close to 3 whereas for 50 concurrent queries, the speed up improved to about 7. When testing 50 random queries we obtained a 50% improvement (for 16 processors the speedup improvement was from 5.8 to 7.5). For 500 short queries we obtain a greater performance improvement because only one or two processors were involved in computing the shortest path due to the small distance between source and destination vertices. Since the 500 short queries were uniformly distributed we obtained better parallel utilization.

II. Idle time and computation time (Figure 4.a) - the idle time of the processors was significantly reduced. This is noticeable in all types of queries and for all processors. E.g., for 500 random queries the idle times of 4 and 16 processors was reduced from 591, and 704 seconds to about 2 and 6.5 seconds, respectively when the number of concurrent queries

Concurrency threshold	processor configuration	COMM	IDLE	COMP	TOTAL	SPEEDUP	efficiency	average response time
50	1x1	0.01	0	3264.64	3264.64	1	100.00%	1592.59
50	2x2	132.16	3.35	1059.39	1194.9	2.73	68.30%	577.47
50	3x3	85.46	25.53	571.84	682.83	4.78	53.12%	333.16
50	4x4	82.59	5.9	347.19	435.69	7.49	46.83%	211.42
20	1x1	0.01	0	3263.2	3263.2	1	100.00%	1594.03
20	2x2	132.45	2.33	1058.32	1193.11	2.74	68.38%	579.50
20	3x3	82.72	22.88	553.43	659.03	4.95	55.02%	332.15
20	4x4	81.94	8.7	343.34	433.98	7.52	47.00%	212.88
5	1x1	0.01	0	3260.74	3260.75	1	100.00%	1587.52
5	2x2	131.29	14.92	1049.57	1195.77	2.73	68.17%	581.27
5	3x3	85.63	41.36	566.75	693.74	4.7	52.23%	337.34
5	4x4	82.25	17.9	340.3	440.45	7.4	46.27%	212.82
1	1x1	0.03	0	3255.88	3255.91	1	100.00%	1591.68
1	2x2	133.42	164.59	1036.51	1334.52	2.44	60.99%	650.08
1	3x3	79.89	318.6	500.1	898.59	3.62	40.26%	431.35
1	4x4	76.26	179.91	305.19	561.36	5.8	36.25%	278.40

Table 1
50 randomly distributed queries (time in seconds)

Concurrency threshold	processor configuration	COMM	IDLE	COMP	TOTAL	SPEEDUP	efficiency	average response time
50	1x1	0.06	0	3090.33	3090.39	1	100%	1517.07
50	2x2	154.32	1.99	1126.33	1282.63	2.41	60.24%	643.62
50	3x3	108.13	8.79	646.46	763.38	4.05	44.98%	379.09
50	4x4	92.31	6.44	352.12	450.87	6.85	42.84%	221.96
20	1x1	0.06	0	3077.31	3077.37	1	100%	1511.51
20	2x2	152.52	1.81	1121.1	1275.42	2.41	60.32%	643.07
20	3x3	107.56	17.77	638.41	763.73	4.03	44.77%	376.07
20	4x4	93.32	11.98	354.47	459.77	6.69	41.83%	228.61
5	1x1	0.06	0	3073.67	3073.73	1	100%	1510.33
5	2x2	152.96	55.41	1105.5	1313.88	2.34	58.49%	666.12
5	3x3	104.05	141.31	592.34	837.69	3.67	40.77%	414.48
5	4x4	90.12	90.89	329.56	510.57	6.02	37.63%	254.44
1	1x1	0.26	0	3085.73	3085.98	1	100%	1516.79
1	2x2	156.24	591.63	1057.34	1805.21	1.71	42.74%	903.78
1	3x3	102.29	935.68	531.35	1569.32	1.97	21.85%	771.71
1	4x4	87.35	704.77	295.33	1087.45	2.84	17.74%	544.72

Table 2
500 randomly distributed short queries (time in seconds)

grew from 1 to 50. Note, that this reduction in idle times increased processor utilization even though some of the work may not have contributed to an increased performance.

III. Average response time (Figure 4.b) - the response time measure shows a significant reduction in the time a user must wait for output. E.g., in the case of 500 random short queries the response time was cut by more than 50% from 544 seconds to 221 seconds for 16 processors.

We find it important to observe that the level of concurrency does not have to be large in order to obtain a significant improvement in performance. Note that, although there is a continued improvement as the number of concurrent queries grows from 5 to 50, it is not as dramatic as from 1 to 5 concurrent queries. Although, this may be somewhat counter intuitive it can be explained when examining the idle time of the processors (e.g., Figure 4(1.a)). The idle time drops 7-10 fold as the number of concurrent queries grows from 1 to 5.

5. Conclusion

In this paper we present a new paradigm of parallelism - concurrent parallelism where both tasks and executions of individual tasks are parallelized. Through experimental results we showed that concurrent parallelism significantly reduces processor idle times and improves over-all performance by about 15%-50%. By using concurrency we were able to obtain efficiencies of up to 86%, 76% and 72% for 4, 9, 16 processors, respectively on the Sunfire, and 88%, 79% and 65% for 4, 9, and 16 processors, respectively on the Beowulf. These efficiencies are much higher than those obtained without concurrency (61.0%, 40%, and 36% for 4, 9, 16 processors, respectively on the Beowulf).

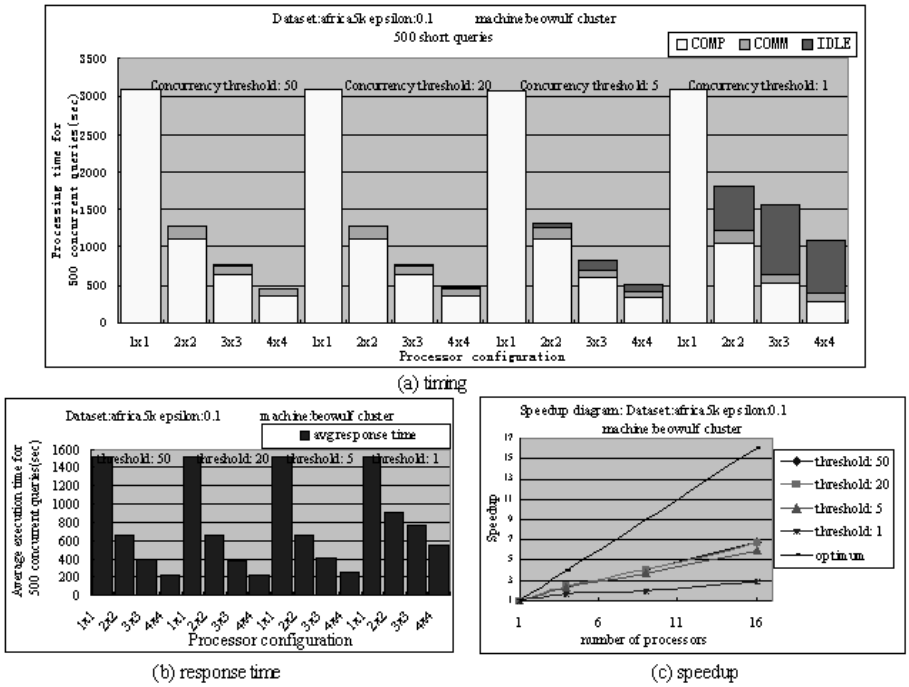


Figure 4. Results of 500 random short distributed queries with different concurrency thresholds.

References

[1] P. Adamson and E. Tick. Greedy partitioned algorithms for the shortest-path problem. *International Journal of Parallel Programming*, 20(4):271–298, 1991.

[2] L. Aleksandrov, M. Lanthier, A. Maheshwari, and J.-R. Sack. An epsilon-approximation algorithm for weighted shortest paths on polyhedral surfaces. In *SWAT: Scandinavian Workshop on Algorithm Theory*, 1998.

[3] L. Aleksandrov, A. Maheshwari, and J.-R. Sack. Approximation algorithms for geometric shortest path problems. In *the 32nd Annual ACM Symposium on Theory of Computing*, pages 286–295, 2000.

[4] L. Aleksandrov, A. Maheshwari, and J.-R. Sack. An improved approximation algorithms for computing geometric shortest paths. In *Foundations of Computation Theory*, pages 246–257, 2003.

[5] L. Aleksandrov, A. Maheshwari, and J.R. Sack. Determining approximate shortest paths on weighted polyhedral surfaces. *Journal of the ACM (JACM)*, 52:25–53, January 2005.

[6] D.P. Bertsekas, F. Guerriero, and R. Musmanno. Parallel asynchronous label-correcting methods for shortest paths. *Jounral of Optimization Theory and Applications*, 88(2):297–320, 1996.

[7] E.W. Dijkstra. A note on two problems in connection with graphs. *Numerical Mathematics*, 1(1):269–271, 1959.

[8] M. Hribar, V. Taylor, and D. Boyce. Choosing a shortest path algorithm. Technical Report CSE-95-004, Northwestern University, 1995.

[9] M. Hribar, V. Taylor, and D. Boyce. Performance study of parallel shortest path algorithms: Characteristics of good decompositions. In *13th Annual conference on Intel Supercomputers User Group*, Albuquerque, NM, 1997.

[10] M. Hribar, V. Taylor, and D. Boyce. Reducing the idle time of parallel shortest path algorithms. Technical Report CPDC-TR-9803-016, Northwestern University, 1998.

[11] M. Lanthier, A. Maheshwari, and J.-R. Sack. Approximating weighted shortest paths on polyhedral surfaces. *Algorithmica*, 30(4):527–562, 2001.

[12] M. Lanthier, D. Nussbaum, and J.-R. Sack. Parallel implementation of geometric shortest path algorithms. *Parallel Computing*, 29:1445–1479, 2003.

[13] J.S.B. Mitchell and C.H. Papadimitriou. The weighted region problem: Finding shortest paths through a weighted planar subdivision. *Journal of ACM*, 38:18–73, January 1991.

[14] D. Nussbaum. *Parallel spatial modelling*. PhD thesis, Carleton University, Ottawa, Canada, 2001.

[15] J.H Reif and Z. Sun. An efficient approximation algorithm for weighted region shortest path problem. In *4th workshop on Algorithmic Foundations of Robotics(WAFR2000)*, pages 191–203, 2000.

[16] J.L. Träff. An experimental comparison of two distributed single-source shortest path algorithms. *Parallel Computing*, 21:1505–1532, 1995.

[17] M. Ziegelmann. *Constrained Shortest Paths and Related Problems*. PhD thesis, Max-Planck Institut für Informatik (submitted at Universität des Saarlandes), Saarbrücken, Germany, 2001.

OpenMP Parallelizations of Viswanathan and Bagchi's Algorithm for the Two Dimensional Cutting Stock Problem *

Gara Miranda Valladares[a], Coromoto León Hernández[a]

[a] Departamento de Estadística, I. O. y Computación,
University of La Laguna, Tenerife E-38271, Spain.
(gmiranda | cleon)@ull.es

1. Introduction

This paper presents two proposals of parallelization of Viswanathan and Bagchi's algorithm to solve the Two-Dimensional Cutting Stock Problem [12]. Both implementations use the skeleton library MaLLBa [1]. This library provides algorithmic skeletons for solving combinatorial optimization problems. Viswanathan and Bagchi's algorithm is based on a best-first search, so it is easily adaptable to different MaLLBa skeleton interfaces. More precisely, to solve the problem the MaLLBa::BnB [6] and MaLLBa::A* [9] skeletons have been instantiated. These skeletons require from the user the implementation of several C++ classes: a `Problem` class defining the problem data structures, a `Solution` class to represent the result and a `SubProblem` class to specify subproblems. Initially, the problem was implemented using the MaLLBa::BnB interface. Due to the dependent generation of subproblems done by the problem algorithm, the distributed parallel solver provided by this interface was not able to afford the resolution of the problem. An ad hoc parallelization was done over the MaLLBa::BnB sequential solver. Then, the problem was also implemented with the MaLLBa::A* interface. This interface provides a sequential and a parallel solver for A* searches. In this case, the interface has the hoped behaviour when the generation of new subproblems depends on other subproblems previously generated. Both parallel implementations use the shared memory paradigm and the OpenMP [10] tool.

The article content will be organized in the following way: First we will introduce the problem and present Viswanathan and Bagchi's algorithm for the problem resolution. Section 3 is dedicated to the description of MaLLBa skeleton classes that implement the problem. The two proposals of parallelization of this problem implementation will be described in detail in section 4. Computational results on a multiprocessor will be shown in section 5. Finally, conclusions and future works are given.

2. Two-Dimensional Cutting Stock Problem

The Two-Dimensional Cutting Stock Problem has lots of applications in many different types of industries. Its formulation is as follows. Consider a surface S with size $L \times W$ made of a certain material. And a set of n different patterns, each one with dimensions $l_i \times w_i$, with an associated profit c_i. Let's b_i the number of available pieces of type i and x_i the number of pieces of type i that have been used. The problem consists in finding the set of patterns and its distribution along the surface S that get a maximum profit and a minimum loss of the material, that is,

*This work has been supported by the EC (FEDER) and by the Spanish Ministry of Science and Technology with contract number TIC2002-04498-C05-05. Also by the Canary Government Project COF2003/022. The work of G. Miranda has been developed under the grant FPU-AP2004-2290.

$$Maximize \sum_{i=1}^{n} c_i x_i \;\; subject\ to \{R\}$$

where R is a set of specific constraints. Depending on the definition of this constraint set, you will have a certain type of Cutting Stock Problem. Anyway, even supposing the simplest constraint set, Cutting Stock Problems are classified as NP-Hard problems [3].

The first formulation of the Cutting and Packing Problem as a Linear Programming Problem was made in 1961 [5]. Since that moment a lot of bibliography about the different definitions of the problem has been appeared. There are many classifications of these problems depending on the number of dimensions, the number of available surfaces and patterns, the shape of the patterns, the orientation, the availability, etc., [4,11]. The solution to the problem has been studied following multiple approximations. Dynamic Programming techniques, Integer Programming methods, Heuristic searches, etc., have been used, [2,3,8]. In 1989, Viswanathan and Bagchi [12] propose an exact algorithm based on a *best-first* search. In [7] Hifi introduces a modification in the calculation of the algorithm bounds.

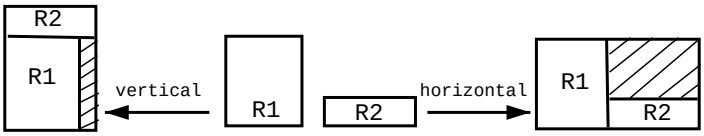


Figure 1. Vertical and horizontal builds

Viswanathan and Bagchi's algorithm considers that any solution to the problem can be obtained by vertical and horizontal combinations of different *builds* of pieces, see Figure 1.

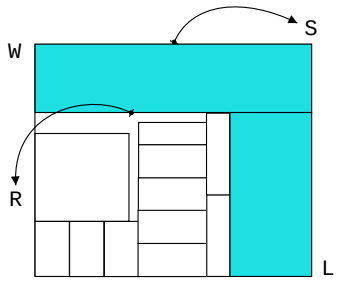


Figure 2. Surface S and build R

At each step, the considered *best-build* is placed in the left bottom corner of the available surface and it is combined with the best builds selected until that moment, see Figure 2.

In order to have a way to know which builds are better and which are worse, it is defined the R *build accumulated profit*, $g(R)$, as the profit sum of all patterns belonging to the build R. Besides, $h(R)$ is defined as the *maximum profit* obtainable from the remaining area of the surface. So, having a certain build R, its *total profit* is defined as: $f(R) = g(R) + h(R)$.

To calculate $f(R)$, the algorithm uses an upper estimation of $h(R)$, denoted as $h'(R)$. To calculate this estimation, functions $U_1(x_R, y_R)$ or $U_2(x_R, y_R)$ are defined:

$$U_1(x_R, y_R) = F(L, W - y_R) + F(L - x_R, W)$$
$$U_2(x_R, y_R) = max\{h_1(x, y), h_2(x, y)\}\ y\ \ U_2(L, W) = 0$$

where F is the function for the Bidimensional Knapsack Problem [5] that is satisfied for all x and y ($x \leq L$ and $y \leq W$) and where $h1$, $h2$ are defined as:

```
begin
  Open := {r_1, r_2, ..., r_n};
  Best := {};
  finished := false;
  repeat
    choose the R rectangle with higher f' value;
    if h'(R) = 0 then
      finished := true;
    else
    begin
      transfer R from Open to Best;
      build all Q guillotine rectangle such as:
        i. Q is an horizontal or vertical build of R with any R' rectangle (included R') from Best;
        ii. Q dimensions are ≤ (L, W);
        iii. Q satisfies all the problem constrains
      put all new Q rectangles in Open with their appropiate g, h' and f' values.
    end;
  until finished;
  Return R as the problem solution;
end
```

Figure 3. Viswanathan and Bagchi's Algorithm

$$F(x,y) = max\{F_0(x,y), F(x_1,y) + F(x_2,y), F(x,y_1) + F(x,y_2)\}$$
$$F_0(x,y) = max\{\{0,c_i\} : l_i \leq x, w_i \leq y, \forall\, i \in 1,\ldots,n\}$$
$$x \geq x_1 + x_2, \quad 1 \leq x_1 \leq x_2$$
$$y \geq y_1 + y_2, \quad 1 \leq y_1 \leq y_2$$
$$h_1(x,y) = max\{U_2(x+u,y) + F(u,y) : 1 \leq u \leq L - x\}$$
$$h_2(x,y) = max\{U_2(x,y+v) + F(x,v) : 1 \leq v \leq W - y\}$$

So, having a certain build R, its *estimated total profit*, $f'(R)$, is defined as: $f'(R) = g(R) + h'(R)$.

Figure 3 shows a pseudocode of *Viswanathan and Bagchi's algorithm*. The presented method follows a scheme very similar to an A* search, using the "total profit" and "estimated total profit" functions described above.

3. The MaLLBa Library

An *Algorithmic Skeleton* must be understood as a set of procedures that compose the structure to use in the development of programs for the resolution of a given problem using a particular algorithmic technique. They provide an important advantage in comparison to a direct implementation of the algorithm from the beginning, not only in terms of code reuse but also in methodology and concept clarity. Skeletons introduce modularity in the design of algorithms. In general, the software that supply skeletons presents declarations of empty classes. The user must fill these empty classes to adapt the given scheme for the resolution of a particular problem.

The problem has been implemented using MaLLBa::BnB and MaLLBa::A* skeletons. User interfaces for both schemes are very similar. The user has to specify similar methods and classes. The main difference between the two skeletons lies in their internal operation. MaLLBa::BnB [6]

implements a Branch and Bound technique over the problem search space. It needs some functions to calculate upper and lower bounds of each subproblem, in order to avoid exploring the hole search space. It explores the tree space, branching each subproblem and bounding the worse branches. When the exploration finishes, the solver returns the best solution found. MaLLBa::A* [9] implements general heuristic searches. An A* search is one of the most complex searches that it is able to do. This kind of search is similar to a best-first search, although it implements some more functionalities. This scheme provides to the user the possibility of selecting very different type of searches by using a configuration class. In a simple configuration it can implement a general Branch and Bound technique. Due to the similarities between both skeleton interfaces, we are going to present the problem implementation into MaLLBa::A*, because it is more complex and offers more configuration alternatives. For this problem the required classes have been defined as shown in Figure 4:

- `Problem`. This class stores the characteristics of the problem to solve. `L` and `W` represent the length and width of the material surface. `n` is the number of different patterns. Vectors `l`, `w`, `p` and `x` are defined to represent the length, width, profit and number of available pieces of each pattern. The `bestBound` field allows to have always updated the value of the current best lower bound (best $g(R)$). In the subproblem generation, this make possible to discard subproblems whose upper bound is worse than the current best lower bound. Table `U_2` contains $U_2(x_R, y_R)$ values for the problem instance. The calculation of U_2 is not exactly the one shown in the algorithm. In this class, the user must specify that the problem to solve is a *maximization* problem.

- `SubProblem`. This class represents a node in the tree or search space. It defines the search for a particular problem and it must contain a field of type `Solution` in which store the (partial) solution. For this problem, it represents a *build* or distribution of pieces. The necessary fields are: `g` holds the accumulated profit of the represented build, `h'` stores the remaining estimated profit, the length `l` and width `w` of the build, vector `n` stores the number of used elements of each type of piece and `sol` has the solution represented by the subproblem. The methods to define in this class are: `initSubProblem(pbm, subpbms)` creates one initial subproblem from each different pattern. `lower_bound(pbm)` calculates the subproblem accumulated profit $g(R)$. `upper_bound(pbm, sol)` calculates the estimated total profit $f'(R)$ of the subproblem. `branch(pbm, subpbms)` generates a set of new subproblems from the current subproblem. `branch(pbm, sp, subpbms)` generates a set of new subproblems obtained from the combination of the current subproblem with a given subproblem sp. When creating new subproblems, the current build has to be combined with all previously analysed subproblems. So, we have to implement last `branch` method and indicate to the skeleton that the generation of subproblems is of $dependent$ type. `ifValid(pbm)` decides if a given subproblem has any success expectatives. It is used in the parallel solver to discard subproblems that are not generated in the sequential resolution. `similarTo(sp)` decides if two given subproblems are similar or not. This method is necessary in order to avoid exploring nodes in cases where similar and better nodes have been analysed before. `betterThan(sp)` decides which of two given subproblems is better. It allows to discard the worst of two similar subproblems. Because of the operation of the algorithm, it can be simply implemented with the skeleton.

- `Solution`. This class defines how to represent the solutions. Vectors `x` and `y` represent the position of the bottom left corner of each pattern on the surface and `pattern` contains the sequence of pieces that make up the build.

```
requires class Problem        // Two-Dimensional Cutting Stock Problem
{
   long L;                    // Surface length
   long W;                    // Surface width
   long n;                    // Number of different patterns
   vector<long> l;            // Patterns length
   vector<long> w;            // Patterns width
   vector<long> p;            // Patterns profit
   vector<long> x;            // Number of available elements for each pattern
   Table U2;                  // Dynamic programming table
   Number bestBound;          // Problem best current bound
   ...
}

requires class Solution
{
   vector<long> x;         // X coordinate (position) of the solution patterns
   vector<long> y;         // Y coordinate (position) of the solution patterns
   vector<long> pattern;   // Sequence of patterns that compose the solution
   ...
}

requires class SubProblem
{
   Number g;               // Accumulated profit of the subproblem
   Number h;               // Maximum obtainable profit of the remaining area
   long l;                 // Length of the current build
   long w;                 // Width of the current build
   vector<long> n;         // Number of elements used of each pattern
   Solution sol;           // Solution represented by the subproblem
   ...
```

Figure 4. MaLLBa classes for the 2D Cutting Stock Problem

Once all these specifications for the problem have been done, MaLLBa::A* sequential solver will work in the following way. First, all initial subproblems are created. Initial subproblems are inserted into the open list. For this problem, subproblems must be always inserted into open by order, from higher upper bound to lower. At each iteration, the first subproblem in the search (open) list is removed. If no other similar and better subproblem has been analysed before, it is branched and inserted into the list of best subproblems. All new subproblems or builds are generated by combining (horizontally and vertically) the best current build with all builds previously expanded (builds in best list). The new created subproblems that verify the problem constrains, are inserted into the search list and the value of the best current bound is updated. Last step is repited until the best current subproblem is a total solution, that is, no more profit is obtainable from the remaining material. For its operation, MaLLBa::A* skeleton uses two linked lists: open and best. The open list contains all the nodes generated but not expanded yet and the best list contains the best expanded nodes.

The MaLLBa skeleton provides to the user the following classes:

- `Setup`. This class is used to configurate all the search parameters and skeleton properties. The user can specify if the best list is needed, the type of insertions to do into open, the type of subproblems generation (dependent or independent), if it is necessary to analyse or not similar subproblems, if search over all the space or if stop when the first solution is found, etc.

- `Solver`. Implements the strategy to do: Branch and Bound, Heuristic searches, etc. Usually, each skeleton provides several solvers. Some of them are sequential and other are parallel.

```
 1 void SubProblem::branch(Problem& pbm, vector<SubProblem*>& subpbms) {
 2   vector<SubProblem*> aux_subpbms;
 3   SubProblem *sp, *new_spV, *new_spH;
 4   bool validV, validH;
 5   long size;
 6   ...
 7   // Insert current subproblem into the list of best subproblems
 8   sp = new SubProblem(*this);
 9   pbm.bs.push_back(sp);
10   size = pbm.bs.size();
11   #pragma omp parallel for private(new_spV, new_spH, validV, validH)
12   for (long i = 0; i < size; i++) {  // Generate all new builds
13     new_spV = new SubProblem();
14     new_spH = new SubProblem();
15     validV = validH = true;
16     generateSpbms(*sp, *(pbm.bs[i]), new_spV, new_spH, &validV, &validH, pbm);
17     // Insert only the valid subproblems
18     if (validV) {           // Vertical build
19       new_spV->h = pbm.calculateBound(*new_spV);
20       if (new_spV->g > pbm.bestBound) {
21         #pragma omp critical (bB_update)
22           pbm.bestBound = new_spV->g;
23       }
24       #pragma omp critical (push_sp)
25         aux_subpbms.push_back(new_spV);
26     } else delete (new_spV);
27     if (validH) {           // Horizontal build
28       new_spH->h = pbm.calculateBound(*new_spH);
29       if (new_spH->g > pbm.bestBound) {
30         #pragma omp critical (bB_update)
31           pbm.bestBound = new_spH->g;
32       }
33       #pragma omp critical (push_sp)
34         aux_subpbms.push_back(new_spH);
35   } } else delete (new_spH);
36   // Delete subproblems that will not get to an optimal solution
37   size = aux_subpbms.size();
38   #pragma omp parallel for
39   for (long i = 0; i < size; i++) {
40     if ((aux_subpbms[i]->g + aux_subpbms[i]->h) >= pbm.bestBound) {
41       #pragma omp critical (push_sp)
42         subpbms.push_back(aux_subpbms[i]);
43     } else delete (aux_subpbms[i]);
44 } }
```

Figure 5. Ad hoc parallelization with MaLLBa::BnB

4. Parallel Schemes

The first aproximation followed to solve the presented problem was made into MaLLBa::BnB skeleton. MaLLBa::BnB skeleton provides a structure with the representation of the search space and allows to do a best-first search. But it does not provide any mechanism to store the expanded nodes. So, the difference with the implementation described befored is that in this case, the user must program the way of storing the explored nodes. At each branch, the user has to update the close or best list and do the combination of the current subproblem with all the subproblems expanded before.

Taking into account the necessary computational effort to do the generation of subproblems, it is first proposed the parallelization of the most hard loops in the `branch` method. This parallelization is done by the user using OpenMP (see Figure 5). First, the subproblem generation and verification loop is parallelized (line 11). Each thread does the combination of two builds (horizontal and vertical) and verifies if the new subproblems are valid. The threads must also update the best bound value obtained for the problem. This is a critical operation because `bestBound` is a shared variable (line 21 and line 30). Threads must be also carefully when inserting subproblems into the list of new

subproblems (line 24 and line 33). The loop that deletes the worst subproblems is also parallelized (line 38).

The solver provided by MaLLBa::A* skeleton is based in a shared memory scheme to store the subproblem lists (open and best) used during the search process. Both lists have the same functionality than in the sequential skeleton. The difference is that the lists are now stored in shared memory and it makes possible that several threads can work simultaneously in the generation of subproblems from different nodes. One of the main problems is to hold the open list sorted. The order in the list is necessary to get always the optimal solution. Moreover, if worse subproblems are first branched, the two lists would grow unnecessarily and useless work would be done. By this reason, standard mechanisms for the work distribution could not be used.

The technique applied is based on a *master-slave* model. Before the threads begin to work together, the master generates the initial subproblems. At that moment, the master and the slaves begin their work to solve the problem. At each step, the master extracts the first node of the open list. If it is a solution, the search finishes. In other case, and assuming that the node is inserted in best, next step consists in verifying if there is some one doing its branch or if it had been done before. If the node is still unbranched and nobody is working on it, the master does this work. If the node is assigned, the master must wait until the thread which works on it finishes to generate its subproblems. Once all the node subproblems have been generated, the master inserts them in the open list. Until the master does not notify the end of the search, each slave works generating subproblems from the unexplored nodes of the open list.

In this scheme, some problems appear when different threads are simultaneously trying to modify the same shared variable. By this reason, several mechanisms have been developed in order to guarantee the synchronization of all threads and the consistent view of all shared variables.

5. Computational Results

For the computational study, we have selected some instances from the ones exposed in [7]. The selected problem instances are: $1\_$, $A4$ and $A5$. The experiments have been run on a machine with 4 processors Intel Xeon 1400 MHz and on an Origin 3800. Here the first one are presented.

Table 1 shows execution times and the number of average computed nodes for the executions of the sequential solvers and the parallel solver with 2 and 4 threads in the case of the MaLLBa::BnB and MaLLBa::A* implementations. Times gotten with MaLLBa::A* skeleton are quite lower than the ones obtained with MaLLBa::BnB skeleton. That is because in MaLLBa::BnB implementation the number of computed nodes is higher. In an A* Search the process stops when the first solution is found. But in a Branch and Bound the process must verify, for each branch, if it is necessary to explored it or not. Anyway, the initial parallelization is not very efficient in comparation to its sequential scheme.

	Sequential		2 Threads		4 Threads	
Problem	Comp.	Time	Comp.	Time	Comp.	Time
MaLLBa::BnB						
1_	12979	23,58	13403	30,35	13919	21,61
A4	45033	178,71	45361	211,23	45712	208,91
A5	13668	14,89	13526	31,83	13115	22,43
MaLLBa::A*						
1_	3502	2,78	4136	4,90	4044	3,39
A4	864	75,94	854	9,51	860	7,36
A5	1674	6,17	1510	6,63	1526	3,13

Table 1

6. Conclusions

In this work, we have presented two implementations of the Two-Dimensional Cutting Stock Problem. One implementation is based on a Branch and Bound technique and the other does an A* search. An ad hoc parallelization has been done over the MaLLBa::BnB user code. The MaLLBa::A* skeleton provides a parallel solver that can correctly be used to solve the problem. Both parallelizations have been done with OpenMP.

We have presented computational results obtained for these implementations. As we have shown, the initial parallelization is not efficient because the necessary synchronization needed to updated the shared variables introduces an important overhead. The improvements introduced by the parallel MaLLBa::A* solver are obtained because the number of generated nodes decreases when more threads collaborate in the problem resolution. That is due to the fact that the update of the best current bound is done simultaneous by all threads, allowing to discard subproblems that in the sequential case have to be inserted and explored. Other advantage of this parallel scheme is that the work distribution between the slaves is balanced. Anyway, it is important to consider that the parallelization of an algorithm where the generation of new subproblems depends on all the previous work done, is quite complex.

Actually, we are working to obtain more results with other problems in order to study the behaviour and efficiency of the skeletons in the implementation of different cases. Finally, some work is done in the implementation of a parallel version of the A* scheme based on the message passing paradigm.

References

[1] E. Alba, et al.: MaLLBa: A Library of Skeletons for Combinatorial Optimization. Proceedings of the Euro-Par'02. Springer-Verlag. 2002.

[2] E. Burke and G. Kendall: Applying Simulated Annealing and the No Fit Polygon to the Nesting Problem. Proceedings of the International Conference on Artifi cial Intelligence (IC-AI'99). CSREA Press. 1999.

[3] K. A. Dowsland and W. B. Dowsland: Packing Problems. European Journal of Operational Research. 1992.

[4] H. Dyckhoff: A Typology of Cutting and Packing Problems. European Journal of Operational Research. 1990.

[5] P. C. Gilmore and R. E. Gomory: The Theory and Computation of Knapsack Functions. Operations Research. 1996.

[6] Gonz´alez, J. R., Le´on, C., Rodr´ıguez, C.: An Asynchronous Branch-and-Bound Skeleton for Heterogeneous Clusters. Proceedings of the EuroPVM-MPI'2004. Springer-Verlag. 2004.

[7] M. Hifi : An Improvement of Viswanathan and Bagchi's Exact Algorithm for Constrained Two-Dimensional Cutting Stock. Computer Operations Research. 1997.

[8] S. Maouche and C. Bounsaythip: Optimizing Textile Shape Placement by Tree Genetic Annealing. Proceedings of the Society for Computer Simulation Conference (SCSC'96). 1996.

[9] G. Miranda: Esqueletos Paralelos A*. Aplicaci´on al Problema de Corte Bidimensional. Escuela T´ecnica Superior de Ingenier´ıa Inform´atica. Universidad de La Laguna. 2004.

[10] OpenMP Architecture Review Board: OpenMP C and C++ Application Program Interface. Version 1.0. 1998.

[11] P. E. Sweeney and E. R. Paternoster: Cutting and Packing Problems: A Categorized, Application-Orientated Research Bibliography. Journal of the Operational Research Society. 1992.

[12] K. V. Viswanathan and A. Bagchi: Best-First Search Methods for Constrained Two-Dimensional Cutting Stock Problems. Operations Research. 1993.

A Parallel Adaptive Algorithm to Improve Precision of Time Series Identification

Juan Antonio Gomez Pulido[a], Miguel Angel Vega Rodriguez[a], Juan Manuel Sanchez Perez[a], Jose Maria Granado Criado[a]

[a]Escuela Politecnica, University of Extremadura, 10071 Caceres, Spain

1. Introduction

In this work we present a parallel technique to optimize time series modelling in order to obtain high precision predictions. Also, this technique could be very useful when the high precision mathematical modelling of dynamic complex systems is required. We employ System Identification algorithms, and use recursive least squares processing and ARMAX modelling. After explaining the proposed heuristic (a set of parallel processing units that performs an adaptive algorithm) and the tuning of its parameters, we show the results we have found for several benchmarks. Thus, we demonstrate how the result precision improves.

2. Modeling and Predicting Time Series

In many engineering fields it is necessary to dispose of mathematical models for studying the behaviour of dynamic systems whose mathematical description is not available "a priori". One interesting type of these systems is the Time Series (TS). Time series are used to describe systems in many fields: meteorology, economy, physics, etc. When dealing with TS there is only available a signal under observation; its physical structure is not known. This led us to employ planning System Identification (SI) techniques [1] in order to obtain the TS model. The model precision depends on the assigned values to certain parameters. In this paper we propose a heuristic to adjust these parameters with the aim of improving the precision of the model.

We consider a TS as the description of a simple dynamic system, by means of a sampled signal with period T that is modelled with an ARMAX [2] parametric description (see equation 1: ARMAX model of a TS, where na is the model dimension).

$$y(k) + a_1 y(k_1) + ... + a_{na} y(k_{na}) = 0 \tag{1}$$

Basically the identification consists in determining the ARMAX model parameters a_i (θ in matricial notation) from measured samples $y(k_i)$ ($\varphi(k)$ in matricial notation). Then it is possible to compute the estimated signal $y_e(k)$ (equation 2: estimated value of the TS at k time) and compare it with the real signal y(k), computing the generated error at k time.

$$y_e(k) = [-a_1 y(k-1) - ... - a_{na} y(k-na)] = \varphi^T(k)\theta \tag{2}$$

The recursive estimation updates a_i in each time step k, thus modelling the system (fig. 1). The more sampled data processed, the more precision for the model, because it has more information about the system behaviour history. We consider SI performed by the Recursive Least Squares (RLS) with forgetting factor (λ) algorithm [2]. From the initial conditions, we build $\varphi^T(k)$, and then RLS runs iterations to evaluate y_e. This algorithm is specified by the constant λ, the initial values and the observed samples y(k). There is not any fixed value for λ, even it is used a value between

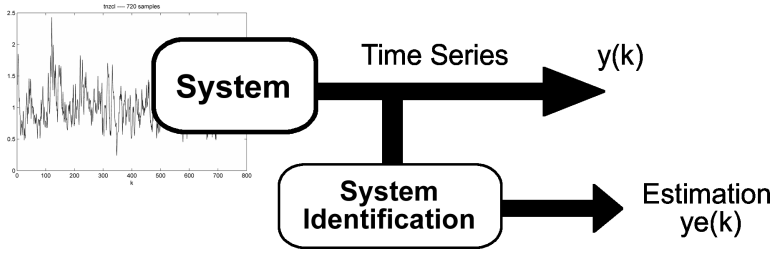


Figure 1. Time series identification

0.97 and 0.995 [3]. The cost function F (see equation 3, where SN is the sample number) is defined as the value to minimize in order to obtain the best precision.

$$F(\lambda) = \sum_{k=k_0}^{k=k_0+SN-1} |y_e(k) - y(k)| \tag{3}$$

The recursive identification is useful for predicting the system behaviour. For example, it could be interesting to know the future behaviour that cannot be experimentally predicted in the prevision of critical or emergency situations. However, for controlling purposes, it is necessary to make a prediction, and for predicting it is necessary to obtain information about the system. This information, acquired by means of the System Identification, consists in elaborating a mathematical model for covering the system behaviour under any working conditions, even under the more extremes.

SI allows finding, in sample time, a mathematical model from which is possible to predict future behaviours. As identification advances in the time, the predictions improve using more precise models. For example, we can compute in sample time the system model and then, with this model to simulate the system future behaviour, forwarding real situations.

3. Parameter Optimization to Improve Prediction Precision

When SI techniques are used, the model is generated "a posteriori" by means of the measured data. However we are interested in the system behaviour prediction in running time, that is, while the system is working and its data are being observed. So, it would be interesting to generate models in running time in such a way that a processor may simulate the system next behaviour.

At the same time, our first effort is to obtain a high model precision (minimal F). System Identification precision is due to several causes, mainly to the forgetting factor λ (fig. 2). Frequently this value is critical for model precision. Other sources also can have less degree of influence (model dimensions, etc), but they are considered as problem definitions, not parameters to be optimized.

On the other hand, it may appear the precision problem when a system model is generated in sample time: If the system response changes quickly, then the sample frequency must be high for avoiding the key data loss in the system behaviour description. If the system is complex and its simulation from the model to be found must be very trustworthy, then the required precision must be very high and this implies a great computational cost. Sometimes the hardware resources do not allow the computational cost in the model generation and processing to be lower than the sample period. We find the trade-off between a high sample frequency and a high precision in the algorithm computation. Adjusting the parameter λ for improving the precision can be conveniently done by means of the parallel technique we present in this work.

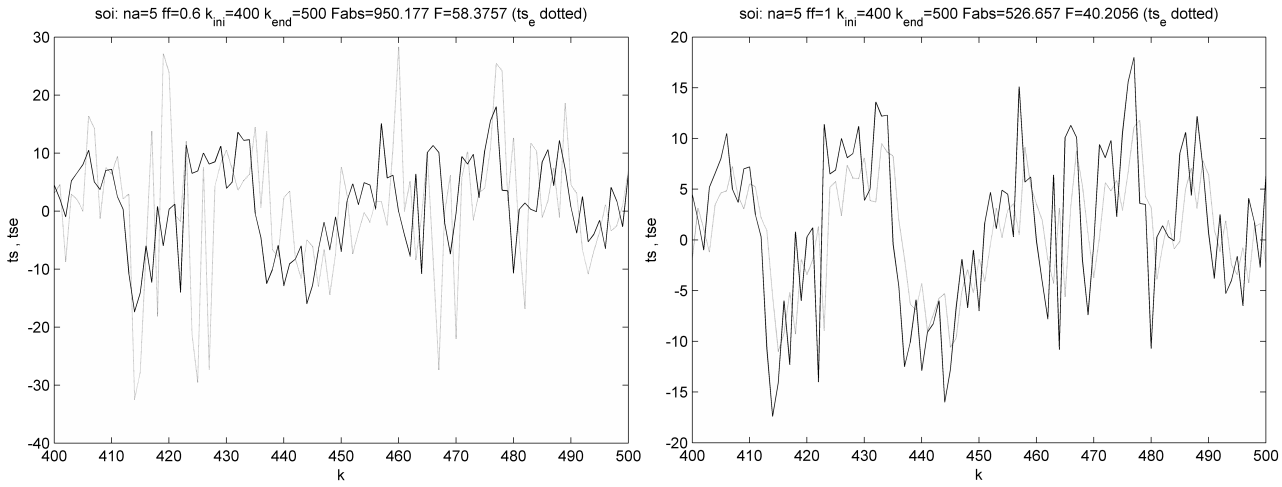


Figure 2. System Identification RLS of benchmark ball with different values for λ parameter. In the left case (λ=0.6) the produced error is greater than in the right case (λ=1)

A consideration prior to the application of the parallel algorithm for optimizing λ is setting the most adequate size of the models to generate in the identification. In other words: the degree of the polynomial expression ARMAX that represents the mathematical formulation of the time series. The notation of this parameter, which from now on we will call dimension of the model, is na.

In the fig. 3 the found results in the analysis performed for several time series benchmarks are shown. These benchmarks have been taken from some time series collections [4][5][6]. For each of them the following experimental procedure has been carried out:

- The results of the cost function F have been evaluated for 200 values of λ comprised to regular intervals in the range (0.9-1.1).
- The best of these obtained results is named Fopt, and written down for the value of the considered dimension of the model.
- This analysis is iterated for 147 different values of na, comprised in the range (3-150). The generated set of Fopt is graphically represented in the fig. 3.

It is deduced from the analysis of the behaviour of the fig. 3 that it can not talk about a general guideline that allows us to determine exactly which is the best dimension for the models. However, we can see the results worsen if the size of the model increases too much. From the data we have obtained here, together with the conclusions extracted from the experimentation carried out in other works [7], we can fix an adequate value of the models size in 15.

In order to find techniques to improve the precision in the time series prediction finding the best λ value, we propose a parallel adaptive algorithm, which is explained in depth in the next section.

4. A Parallel Adaptive Heuristic

In order to find the optimum value of λ, we propose a parallel algorithm that is partially inspired on the concept of artificial evolution [8][9] and also in simulated annealing mechanism [10]. In our algorithm, named PARLS (Parallel Adaptative Recursive Least Squares), the optimization parameter

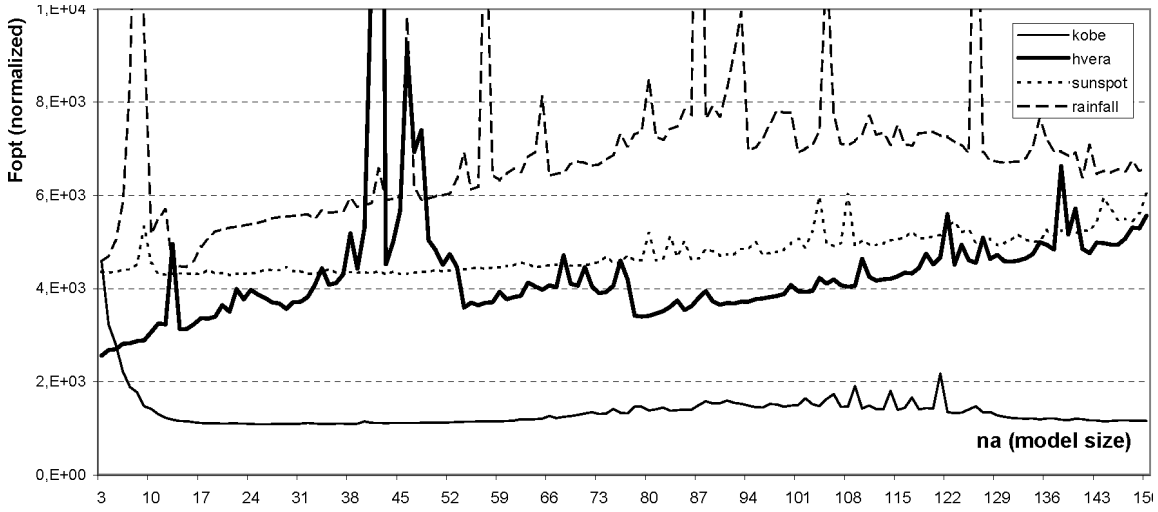


Figure 3. Smaller error (Fopt) in the estimation for 200 λ values vs. model size (ARMAX degree from 3 to 150). This analysis has been made for 4 time series benchmarks of different sizes

R	Generation interval
λc	λ central in R
PHS	Phase samples
PHN	Number of phases
PUN	Number of parallel processing units
TSN	Total number of samples
RED	Reduction factor of R

Table 1
PARLS nomenclature: the more important algorithm's parameters

λ is evolved for predicting new situations during the successive phases of the process. In other words, λ evolves at the same time that improves the cost function performance.

PARLS considers a λ value as a state. Starting on an initial λ value (λc) and an initial R value (the interval of generation where λc is in the middle), a set of λ values is generated covering uniformly the interval R. The λ values generated are equal to the number of parallel processing units (PUN). Each phase of PARLS is an identification loop that considers a given number of sample times (PHS) and the corresponding λ value. We use the nomenclature showed in Table 1.

In each phase, R is reduced dividing itself by the RED factor (the interval limits are moved so the center of interval corresponds with the optimal λ value found in the previous phase), in such a way that the generated set of λ will be more and more near of the previous optimum found. The new set of generated λ values covers uniformly the new R. In each processing unit, during each phase, the cost function F is computed (the accumulated error of the samples that constitutes each phase). From equation 3 we obtain the equation 4 as the cost function for each parallel processing unit.

$$F(\lambda_{PU_X}) = \sum_{k=k_0}^{k=k_0+PHS-1} |y_e(k) - y(k)| \tag{4}$$

At the end of each phase, the best λ is chosen. This is the corresponding value to the lower F. From this λ, new values are generated in a more reduced (new R) interval (see fig. 4). The goal

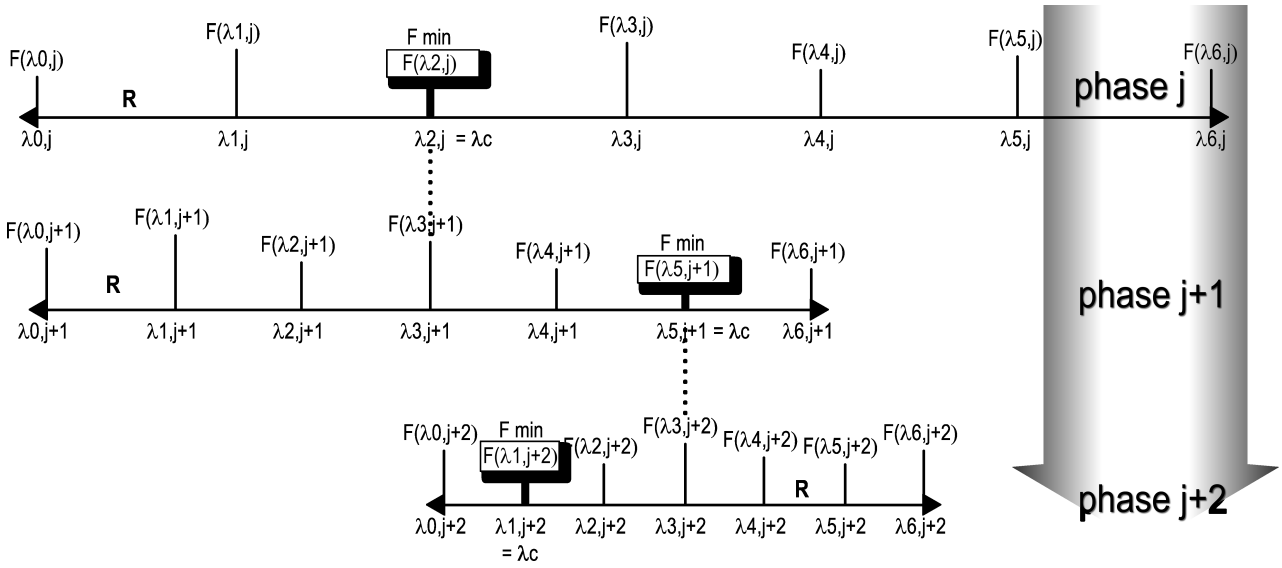


Figure 4. The SI uses different λ values in each phase performed by each processing unit. All the λ values in the same phase running in the processing units are generated in the R interval from the previous phase optimum λ found, corresponding with the smallest computed F

is that the identifications performed by the processing units will converge to optimum λ parameters when a given stop criteria will be achieved. In fig. 5 we can see this parallel architecture what would to help us to know better the PARLS performance.

5. Experimental Results

We consider several criteria for evaluating PARLS performances. All these criteria have been fully checked and tested in order to get a set of better values for parameters and strategies. For example, we have studied strategies as the optimum λ criteria (the λ value that produces a minimum F), the stop criteria (indicating when a processing unit must stop the work), the model generation criteria (how to consider the initial model in the next phase), the optimum F definition (to consider the optimum F as the lowest in all phases or the lowest computed in the present phase), etc.

Another important question is how to establish the initial range of λ values in PARLS search. We have performed several experiments in order to determine the approximated optimal λ for many benchmarks using a lot of RLS computations. In fig. 6 some of these experiments are shown. We can see that there is a distinct optimal λ for each benchmark, but in all cases there is a smooth U-curve that is very useful to the initial PARLS search. We have thus selected as initial searching parameters tuned values λc =1 and R=0.1.

PARLS offers a great variability for its parameters. We have carried out many experiments with a wide set of benchmarks. According to the results we have obtained, we can conclude that there are not common policies for tuning the parameters in such a way that always the best results will be found. But results indicate that there is a set of values for which the results are good. We can thus establish fixed values for PARLS parameters (see table 2) in order to define an unique algorithm applicable to any system.

In the table 3 we show the comparison of found results between RLS search and PARLS heuristic

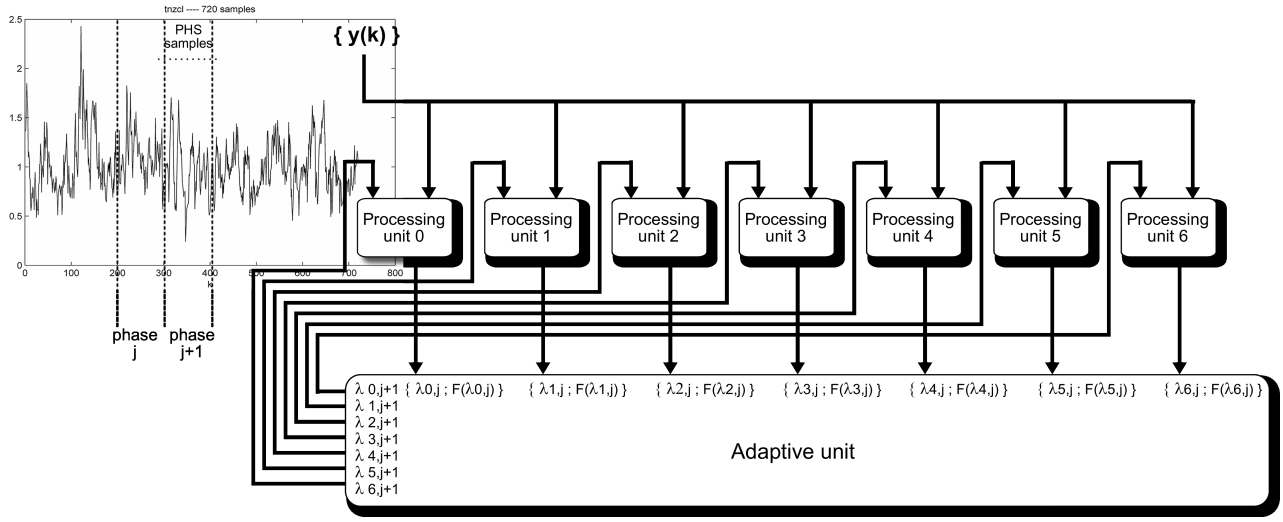


Figure 5. Parallel architecture for the adaptive high precision time series prediction

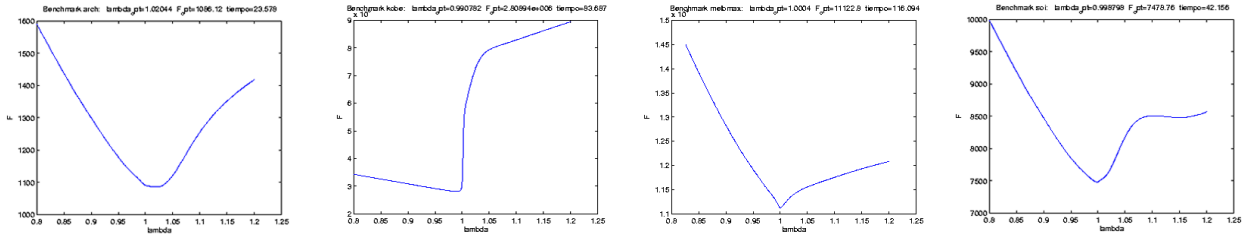


Figure 6. The cost function of six benchmarks calculated by RLS for 500 λ values in the same range (λc=1, R=0.4), using the dimension na=5. We can see a smooth U-curve in all the cases

Parameter	Tuned value
na	5
λc	1
R	0.05
PHN	4
PUN	11
RED	2

Table 2
Main PARLS parameter tuned

Benchmark	TSN	PHS	(a) F for RLS 11	(b) F for PARLS	Better algorithm	PARLS vs RLS 11
kobe	3.048	762	2.80896E+06	2.16053E+06	PARLS	30%
melbmin	3.648	912	7.28124E+03	7.46543E+03	RLS	-2%
melbmax	3.648	912	1.11282E+04	1.10042E+04	PARLS	1%
hvera	1.096	274	2.78909E+03	2.73085E+03	PARLS	2%
arch	1.000	250	1.08611E+03	1.03681E+03	PARLS	5%
f060	4.096	1.024	8.06796E+04	8.10842E+04	RLS	0%
n005	4.096	1.024	9.68213E+04	9.42284E+04	PARLS	3%
o094	4.096	1.024	4.01413E+04	3.87788E+04	PARLS	4%
s059	4.096	1.024	1.23377E+05	1.23183E+05	PARLS	0%
z002	4.096	1.024	3.65868E+04	3.70197E+04	RLS	-1%
soi	1.232	308	7.49888E+03	7.45276E+03	PARLS	1%
eeg01	18.432	4.608	4.24190E+05	6.70278E+04	PARLS	533%
eeg02	18.432	4.608	7.37492E+05	5.79659E+04	PARLS	1,172%
eeg03	15.360	3.840	4.42649E+04	4.29920E+04	PARLS	3%
eeg04	8.192	2.048	3.03256E+04	2.86598E+04	PARLS	6%
tnzcl	720	180	1.02752E+02	1.07755E+02	RLS	-5%
rainfall	3.652	913	7.54512E+04	6.93007E+04	PARLS	9%

Table 3
Comparison of results between RLS search and PARLS. The better algorithm is PARLS for the greater part of the benchmarks, and in the other cases the PARLS results is closely near to RLS search (see the % of PARLS better than RLS)

for several benchmarks. In all cases the same tuned parameters has been used (na=5, PUN=11, λc=1, R=0.05, RED=2, PHN=4). In (b) the results of 11 RLS identifications with their corresponding 11 equidistant in R values of λ are shown, and in (b) the PARLS results are displayed too. The computational effort of 11 RLS identifications is almost equal to PARLS cost with 11 processing units, so both results can be compared. With these tuned parameters, PARLS finds better results in the greater part of benchmarks. However, and this is important, for the other benchmarks, in all cases the difference of F oscillates between 2% and 5%. That is, PARLS improves or holds the results found with RLS with the same computational effort.

6. Conclusions and Future Works

With the tuned parameters and benchmarks considered, PARLS finds better results in the greater part of experiments. We can say the parallel adaptative heuristic PARLS offers a good performance, and this encourages us to follow this research. Also, a neural network implementation of the parallel processing units has been developed [11] in order to evaluate new computational costs, with good results. Now, our present effort is oriented to achieve a PARLS synthesis on reconfigurable hardware systems to improve the global efficiency [12], because it could accelerate the algorithm computation.

Also, we are now developing a parallel genetic algorithm (see fig. 7) as a new heuristic to optimize λ value. The obtained results up today say us this is a very interesting research field to explore.

7. Acknowledgements

This work has been developed thanks to TRACER project [13] (TIC2002-04498-C05-01, Ministerio de Ciencia y Tecnologia, Spain, 2002-2005) working in it the following spanish universities: Universidad de Extremadura, Universidad de Malaga, Universidad Politecnica de Cataluna, Universidad de La Laguna and Universidad Carlos III de Madrid.

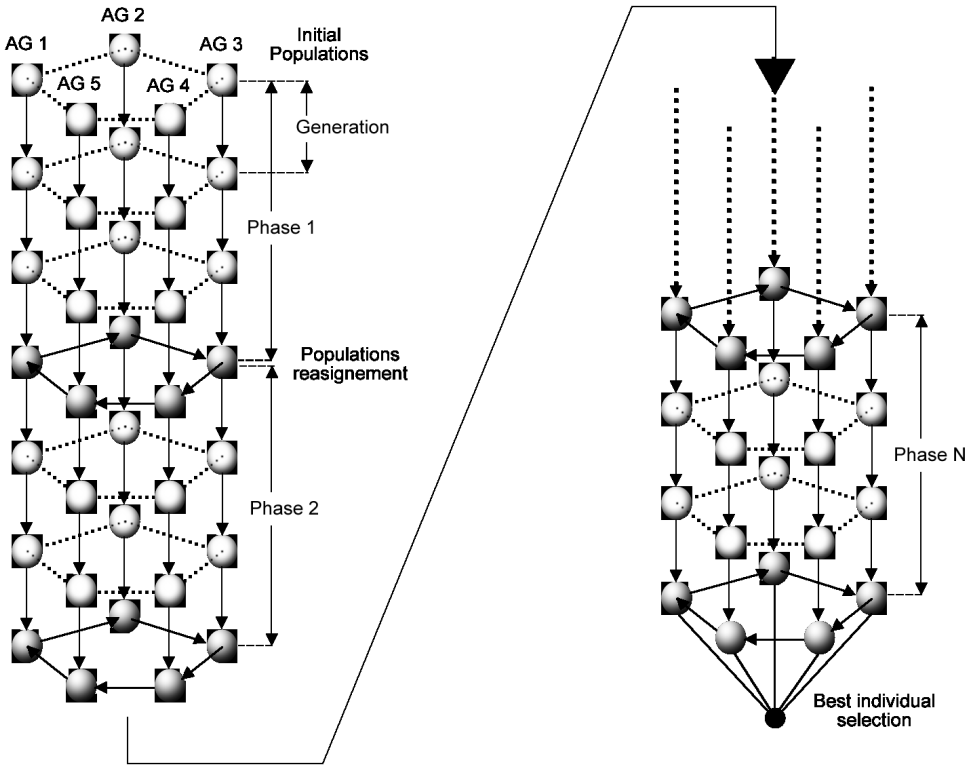


Figure 7. The scheme of the proposed parallel genetic algorithm

References

[1] T. Soderstrom et al.: System Identification. Prentice-Hall. 1989.
[2] L. Ljung: System Identification. Prentice-Hall. 1999.
[3] L. Ljung: System Identification Toolbox. The Math Works Inc. 1991.
[4] A Database for Identification of Systems: http://www.esat.kuleuven.ac.be/sista/daysi
[5] Royal Observatory of Belgium, Brussels: Sunspot Data Series. 2004.
[6] NOAA's National Geophysical Data Center: Sunspot numbers. 2004.
[7] J. A. Gomez, M.A. Vega, J.M. Sanchez: Parametric Identification of Solar Series based on an Adaptive Parallel Methodology. J. Astrophys. Astr. 26, pp 1-13. 2005.
[8] D. E. Goldberg: Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley. 1989.
[9] I. Rechenberg: Evolutionsstrategie: Optimierung tech. sys. nach prinzipien der evolution. Formmann-Holzboog Verlag. 1973.
[10] S. Kirkpatrick, C. Gelatt., M. Vecchi: Optimization by Simulated Annealing. Science, 220, 4598, pp. 671-680. 1983.
[11] J.A. Gomez, J.M. Sanchez, M.A. Vega: Using Neural Networks in a Parallel Adaptive Algorithm for the System Identification Optimization. Lecture Notes on Computer Sciences 2687. Springer-Verlag, pp. 465-472. 2003.
[12] Gomez, J. et al.: Diseno de un coprocesador reconfigurable para Identificacion de Sistemas. II Jornadas sobre Computacion Reconfigurable. Univ. de Granada (Spain), pp. 221-226. 2002.
[13] TRACER: Tecnicas de Optimizacion Avanzadas para Problemas Complejos Estocasticos. http://tracer.lcc.uma.es. 2002.

Towards Robustness in Parallel SAT Solving

Wolfgang Blochinger[a]

[a]Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, Sand 14, D-72076 Tübingen, Germany

This paper deals with a novel method for parallel Boolean satisfiability (SAT) solving. Our approach focuses on improving robustness, which plays a key role in enabling practical usability of parallel SAT. Specifically, we adaptively induce competition parallelism into parallel computations based on exploratory decomposition in order to control work-anomalies. We present initial performance measurements indicating the usefulness of our approach.

1. Introduction

The Boolean satisfiability (SAT) problem consists of finding a variable assignment for a Boolean formula F such that F evaluates to TRUE, resp. of proving that for F no satisfying variable assignment exists. Within the class of constraint satisfaction problems, SAT is increasingly gaining importance, since considerable research has been carried out on efficiently encoding real-world problems as SAT instances. Prominent application domains of SAT are electronic design automation (EDA) [19], software-verification [8], scheduling [10], ai planning [16], cryptography [20], and configuration of complex systems [17,21]. Today, SAT solving is recognized as a universal tool for tackling hard problems. Despite significant performance improvements of SAT solving algorithms realized in the last decade, there still exist unsolved SAT instances in all major application fields. For example, the constantly increasing complexity of chip designs is delivering extremely hard SAT instances which are far out of range of state-of-the-art sequential solvers.

Basically, parallel SAT solving is capable of substantially speeding-up the solving process. But a severe limitation of current parallel SAT solvers is that they often exhibit poor robustness. This means that for solving SAT instances with similar complexity, parallel SAT solvers achieve considerably different parallel efficiencies. Even for iterated parallel runs of the same problem instance, often totally different speedups can be observed. This behavior results from work-anomalies, where the total amount of work (in terms of the search space of variable assignments to be tested) differs significantly between sequential and parallel runs of a SAT problem instance.

Improving robustness is crucial for the practical applicability of parallel SAT solving. In this paper, we identify conditions which lead to work-anomalies and study a novel approach to parallel SAT solving which aims at improving robustness of parallel SAT solvers by adaptively combining different forms of parallelism.

The rest of the paper is organized as follows. In Section 2 we give a brief overview of state-of-the-art SAT solving algorithms. Section 3 presents our approach for improving robustness of parallel SAT. In Section 4 we report on initial performance measurements. Section 5 discusses related work.

2. SAT Solving

2.1. Basic Definitions

We consider Boolean formulae in conjunctive normal Form (CNF). In CNF, a formula is composed of conjunctions ($\wedge$) of *clauses*. A clause is the injunction ($\vee$) of one or more *literals*, and a literal is

```
boolean DPLL() {
  while(true) {
    if (decide()) {                              // decision
      while(deduce()==CONFLICT) {                // deducing
        if (current_level==0) {      // top-level conflict
          return false;                    // unsatisfiable
        } else {
          new_level=analyze_conflicts();         // learning
          back_track(new_level);            // backtracking
        }
      }
    } else {                      // all variables assigned
      return true;                             // satisfiable
    }
  }
}
```

Figure 1. DPLL Algorithm with Dynamic Learning and Conflict Driven Backtracking

a variable or the complement of a variable. Consider the following Boolean formula:

$$F = (x_1 \vee x_3) \wedge (x_2 \vee \overbrace{\overline{x_3}}^{literal}) \wedge (\underbrace{\overline{x_1} \vee \overline{x_2} \vee \overline{x_3}}_{clause}) \wedge x_3$$

The variable assignment $x_1 \rightarrow$FALSE, $x_2 \rightarrow$TRUE, $x_3 \rightarrow$TRUE represents a satisfying assignment of F. A fundamental property of a formula in CNF is that it is satisfiable iff. in each clause at least one literal evaluates to TRUE. If for a clause all but one literals have already been assigned to FALSE, the remaining literal must be assigned to TRUE in order to satisfy the clause. Such clauses are called *unit clauses*. A situation when all literals of a clause are assigned to FALSE is called a *conflict*, and the clause is called a *conflicting clause*.

2.2. DPLL SAT Solving Algorithm

The original Davis-Putnam-Logemann-Loveland (DPLL) SAT solving algorithm [12,11] still represents the algorithmic framework of modern complete SAT solvers, but has been significantly enhanced by sophisticated heuristics for pruning the search space of variable assignments to be tested. Most beneficial advances could be achieved by employing *dynamic learning* and *conflict driven backtracking* techniques [18]. Figure 1 shows the basic structure of the DPLL algorithm incorporating these heuristics. We will restrict the following discussion of the DPLL algorithm to a top-level treatment. A more detailed explanation can be found in [23].

Basically, the DPLL algorithm performs a backtrack search process. Partial variable assignments are speculatively extended to find a satisfying assignment. The procedure `decide()` determines according to a *decision heuristics* [14] which unassigned variable should be chosen next to extend the current partial variable assignment. Each decision is recorded on an *assignment stack* along with an associated *decision level*. The decision level of the first decision is 1. The procedure `deduce()` infers additional assignments that are logical consequences of the current partial variable assignment using a technique called *unit propagation*: After making a new decision, some clauses may have become unit clauses implying new assignments. Deduced assignments are called *implications* and are also recorded on the assignment stack at the current decision level. Unit propagation terminates

when either no unit clauses exist or a conflict occurs. In the first case, a new decision is made inducing the next decision level. In the second case, the procedure `analyze_conflicts()` is invoked which constitutes the core of modern SAT solvers. It performs two tasks:

- **Dynamic Learning**: A new clause called *lemma* is constructed by analyzing the reasons for the current conflict. A lemma reflects a minimal subset of the current assignments that implies the conflict. When added to the input formula, a lemma prevents the further search process from reproducing the same conflict in other regions of the search space. Since the underlying mechanism for constructing lemmas is resolution, adding a lemma to the input formula does not affect the correctness of the DPLL algorithm. Original clauses and lemmas constitute the *clause database*.

- **Conflict Driven Backtracking**: By construction, a lemma is initially a conflicting clause. The backtracking level is determined as the lowest level at which the lemma becomes a unit clause. Note that at this level the current conflict is also resolved.

The procedure `back_track()` releases all assignments recorded on the assignment stack up to the computed backtracking level. The newly added lemma, which is now a unit clause, takes the search to a new direction. When backtracking reaches decision level 0, the current lemma forces an assignment at level 0, called a *top-level assignment*. The corresponding conflict cannot be resolved by releasing any non top-level assignments. Thus, top-level assignments are a necessary condition for the formula to be satisfiable and are fixed for the further search process. If all variables have been assigned without a conflict, the input formula is satisfiable. Unsatisfiability of the input formula is proven when a *top-level conflict* occurs (i.e. a conflict at decision level 0), since it cannot be resolved by releasing assignments.

In order to prevent the search process from getting stuck in a futile part of the search space, often a technique called *search restarts* [1] is applied. Here, the search process is periodically cancelled by backtracking to level 1 and restarted keeping some results (typically lemmas) of the previous run.

3. Improving Robustness of Parallel SAT Solving

3.1. Decomposition vs. Competition

Basically, there are two approaches to the parallelization of heuristic search problems of the kind of SAT: *decomposition* and *competition*.

The decomposition approach splits the whole search space of variable assignments into disjoint subspaces to be treated in parallel (*exploratory decomposition*). Due to the highly irregular structure of the search space, particularly of real-world SAT instances, dynamic problem decomposition and consequently dynamic load balancing become inevitable.

The competition approach is based on the property that the effect of search heuristics can vary considerably for different problem instances and cannot be predicted. Here, parallelism is exploited by concurrently executing sequential SAT solvers on the same problem instance, each pursuing a different search strategy until the problem is solved by one of the solvers.

3.2. Robustness of Exploratory Decomposition

For reasons detailed subsequently, applying exploratory decomposition for parallel SAT solving can cause considerable work-anomalies and thus limited robustness. The dynamic learning process of modern SAT solvers relies on accumulated knowledge (in the form of lemmas) which is continuously deduced during the solving process. Employing exploratory decomposition techniques on a

distributed-memory architecture results in a (partially overlapping) partition of the clause database consisting of several distributed clause databases, one on each node. All clause databases comprise the problem clauses and also locally deduced lemmas. Since dynamic learning can considerably prune the search space, it is crucial to exchange lemmas among the different clause databases, establishing a distributed learning process. Due to the high irregularity of SAT, a synchronous approach for exchanging lemmas among the nodes (for example employing an SPMD style all-to-all broadcast) would cause significant processor idling. Moreover, the total amount of deduced knowledge increases with the number of nodes, making a total exchange approach highly unscalable. Consequently, an asynchronous and selective method of communicating lemmas is more appropriate for realizing a distributed learning process. In [4], we proposed a corresponding scheme that uses mobile agents for gathering and exchanging pertinent lemmas among the distributed clause databases. But with asynchronous communication it is virtually impossible to make the deduced knowledge available on every node in a way such that the resulting (parallel) search space remains exactly identical to the sequential search space. In some cases, even minimal initial differences may rapidly lead to totally different search spaces causing a considerable potential of work-anomalies.

3.3. Adaptive Competition

In order to overcome the identified limitations of exploratory decomposition for parallel SAT solving, we propose to combine decomposition and competition parallelism in an adaptive fashion. Competition parallelism is employed when a particularly hard region of the search space is encountered (which may not be present in sequential program runs). The rationale behind this combined approach is that decomposition is able realize speedup while competition can increase robustness. However, pure decomposition can lead to poor robustness, while pure competition is not able to deliver speedup (over the optimal search strategy). Specifically, our approach to parallel SAT solving starts with exploratory decomposition and adaptively induces competition parallelism when the solving process of a particular subproblem doesn't make sufficient progress.

A subproblem is represented by a corresponding assignment stack. For carrying out dynamic problem decomposition, we apply the guiding path technique [22]. It splits the search space of a subproblem by generating two assignment stacks which define disjoint subspaces (see Figure 2). For dynamic load balancing we employ a distributed task-pool model. Splitting is initiated if the size of the local task-pool falls below a given threshold. A randomized work stealing scheme is used to transfer tasks between the task-pools.

In order to steer the transition from decomposition to competition parallelism a *transition heuristics* is employed which assesses the progress of the solving process of an individual subproblem and decides when to switch the treatment of the subproblems to competition parallelism. Parameters of the solving process that can be considered by a transition heuristics are e.g. the number of unassigned variables, the number of conflicts, or the number of splitting operations the subproblem has already been involved in. An example of a concrete transition heuristics is given in the next section.

When a transition is initiated, the respective subproblem is additionally treated using (one or more) different search heuristics. Also further decomposition of the subproblem is disabled. For current SAT-solving algorithms, examples of possible competition disciplines are decision heuristics, lemma construction methods, or clause database management strategies.

4. Experimental Evaluation

In order to test the usefulness of our approach, we carried out performance measurements on a 15 node cluster which was equipped with 2.6 GHz Intel Xeon processors and 2 GB of main memory per

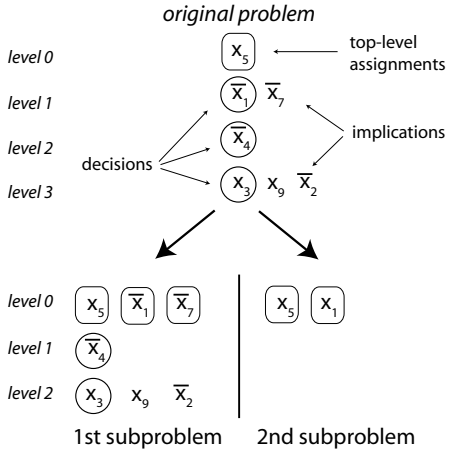


Figure 2. Search Space Splitting

node. All nodes were connected by a GigaBit Ethernet network. Our prototypical implementation is based on the sequential SAT Solver MiniSAT [13]. For the subsequently presented run-time measurements, we chose the grid-10-20 and the longmult15 benchmark from the SAT solver competition benchmark suite.

In our experiments, competition has been established by employing different decision heuristics. Modern SAT solvers typically use a small amount of randomization within the decision procedure, e.g. in MiniSAT 2% of the decisions are made randomly. In order to preserve reproducibility, a fixed random seed is used. We take advantage of this feature to generate different decision procedures by using different (fixed) random seeds, one for each node. For the grid-10-20 benchmark, the sequential run-times for the resulting 15 different decision heuristics ranged from 160.0 to 5142.9 seconds, for the longmult15 benchmark from 159.6 to 300.0 seconds. We choose randomization for establishing competition, because it ensures an unbiased evaluation of the proposed approach. For a later production use, more specific competition disciplines might deliver even more distinct results.

Our transition heuristics is based on a limit of the number of conflicts encountered until the treatment of a subproblem is switched from decomposition to competition. Every node maintains a corresponding limit value which is initialized by the number of variables of the considered SAT instance as an estimate of the problem size. Every time a transition operation takes place, the corresponding limit is increased by a factor of 2 in order to adapt to the total computation time. Additionally, a transition is only carried out, if the considered subproblem has been part of at least one splitting operation. This ensures, that the computation will never degenerate to pure competition.

Figure 3 shows the results of 100 program runs each for employing adaptive competition and pure decomposition. The given sequential run-times represent the measured times of the best performing decision procedure. The run-times for pure decomposition result from parallel runs using the best performing decision procedure.

The results for the longmult15 benchmark show for both approaches nearly identical run-times and also only minimal dispersion. Thus, for this problem instance, decomposition does not induce work-anomalies. However, adaptive competition does not increase the run-time in this situation.

For the grid-10-20 benchmark, the run-times of the decomposition approach exhibit a significant spread indicating a high degree of work-anomalies. Even run-times greater than the sequential time (slow-downs) can be observed. For the program runs which employ adaptive competition in almost all cases considerable speedups could be realized.

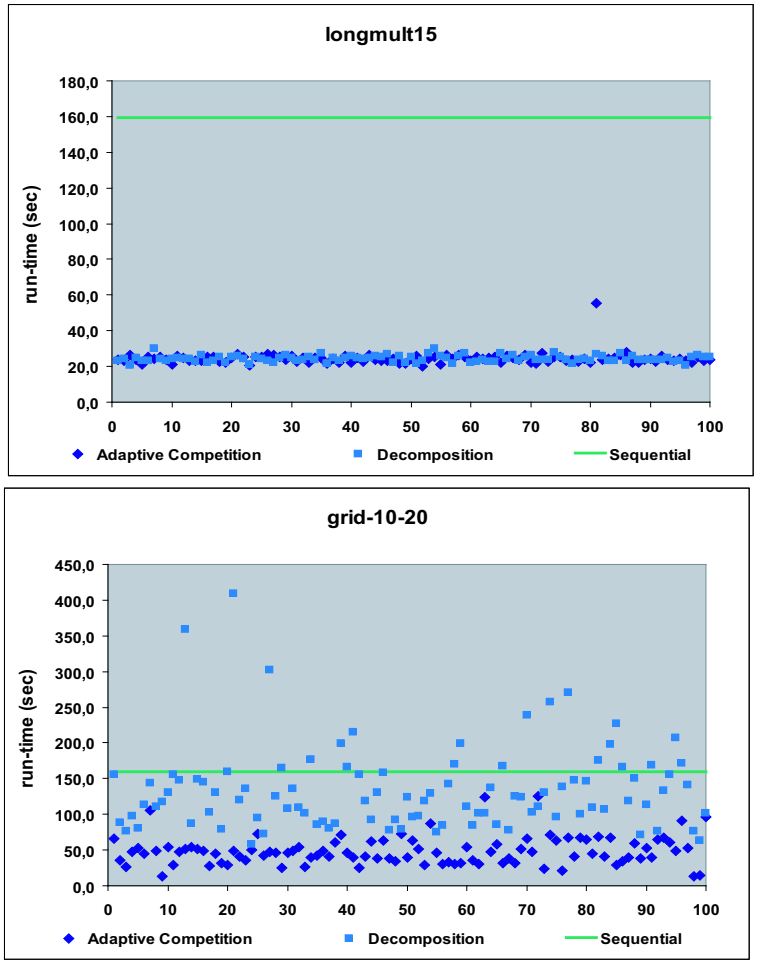


Figure 3. Results of Performance Measurements

5. Related Work

In this section, we discuss other approaches to parallel SAT solving. We restrict our treatment to parallel SAT solvers that are designed for general-purpose parallel hardware. Böhm and Speckenmeyer proposed a parallel SAT solver specifically designed for Transputer systems [7]. This work focuses on studying efficient load balancing techniques for d-dimensional mesh network-topologies in the context parallel SAT solving. Zhang's PSATO [22] is a distributed parallel SAT solver targeted for networks of workstations. PSATO introduced the *guiding path* technique for exploratory problem decomposition, taking advantage of the decision heuristics of sequential solvers for splitting the search space. PSATO is based on external parallelization of the sequential solver SATO. The parallel solver PSatz [15] by Jurkowiak *et al.* is a parallel variant of the sequential solver Satz. Concerning the parallelization approach, PSatz is very similar to PSATO, but uses work-stealing techniques for load-balancing. The parallel SAT solver PaSAT by Blochinger *et al.* [4] focuses on establishing an efficient distributed parallel learning process between the nodes of a cluster or a network of workstations. Also cross-fertilization of different kinds of heuristics is addressed [5]. PaSAT has been successfully employed in an industrial application [3] from the field of automotive

product configuration. It is based on the parallel platform DOTS [2]. GridSAT [9] developed by Chrabakh and Wolski is a parallel SAT solver targeted for parallel resources that are managed by the Globus grid platform. It employs grid technology to make reservations of appropriate dedicated resources (like compute clusters) for executing a parallel SAT solver based on the sequential solver zChaff. ZetaSAT [6] by Blochinger *et al.* is a framework for parallel SAT solving on Desktop Grids, that allows to use idle cycles of desktop computers for solving SAT instances. It is based on the ZetaGrid Desktop Grid platform and uses parallel versions of the SAT solvers zChaff and MiniSAT as solving engines.

All discussed parallel SAT Solvers exclusively apply exploratory decomposition techniques for realizing parallelism. PaSAT and GridSAT additionally establish a distributed learning process. PaSAT employs mobile agents to continuously exchange lemmas among the nodes executing the search process. GridSAT transfers a part of the clause database to newly created tasks when a search space split is performed.

6. Conclusion and Future Work

In this paper, we presented a novel approach to parallel SAT solving that adaptively employs competition parallelism in order to increase the robustness of the parallel solving process. To some extend, the presented approach can be regarded as a refinement of the search restart technique of sequential SAT solvers by using parallelism. But in contrast to search restarts, with adaptive competition work already carried out is not discarded at some point but enters into competition with other strategies.

Our future work will include investigations on sophisticated transition heuristics, other cross-fertilization issues of decomposition and competition parallelism, and on specific load balancing methods.

References

[1] L. Baptista and J. P. Marques-Silva. Using randomization and learning to solve hard real-world instances of satisfiability. In *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming (CP)*, 2000.

[2] Wolfgang Blochinger, Wolfgang Küchlin, Christoph Ludwig, and Andreas Weber. An object-oriented platform for distributed high-performance Symbolic Computation. *Mathematics and Computers in Simulation*, 49:161–178, 1999.

[3] Wolfgang Blochinger, Carsten Sinz, and Wolfgang Küchlin. Parallel consistency checking of automotive product data. In G. R. Joubert, A. Murli, F. J. Peters, and M. Vanneschi, editors, *Proc. of the Intl. Conf. ParCo 2001: Parallel Computing – Advances and Current Issues*, pages 50–57, Naples, Italy, 2002. Imperial College Press.

[4] Wolfgang Blochinger, Carsten Sinz, and Wolfgang Küchlin. Parallel propositional satisfiability checking with distributed dynamic learning. *Parallel Computing*, 29(7):969–994, 2003.

[5] Wolfgang Blochinger, Carsten Sinz, and Wolfgang Küchlin. A universal parallel SAT checking kernel. In Hamid R. Arabnia and Youngsong Mun, editors, *Proc. of the Intl. Conf. on Parallel and Distributed Processing Techniques and Applications PDPTA 03*, volume 4, pages 1720–1725, Las Vegas, NV, U.S.A., June 2003. CSREA Press.

[6] Wolfgang Blochinger, Wolfgang Westje, Wolfgang Küchlin, and Sebastian Wedeniwski. ZetaSAT – Boolean satisfiability solving on desktop grids. In *Proc. of the IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2005)*, Cardiff, UK, 2005.

[7] M. Boehm and E. Speckenmeyer. A fast parallel SAT-solver – efficient workload balancing. *Annals of Mathematics and Artificial Intelligence*, 17(3-4):381–400, 1996.

[8] Sagar Chaki, Edmund M. Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering*, 30(6):388–402, June 2004.

[9] Wahid Chrabakh and Rich Wolski. GridSAT: A Chaff-based distributed SAT solver for the grid. In *Proc. of Supercomputing 03*, Phoenix, Arizona, USA, 2003.

[10] James M. Crawford and Andrew B. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *Proc. of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, volume 2, pages 1092–1097, Seattle, Washington, 1994. AAAI Press/MIT Press.

[11] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[12] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[13] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing: 6th International Conference, SAT 2003*, volume 2919 of *LNCS*, pages 502 – 518, Santa Margherita Ligure, Italy, 2003. Springer Verlag.

[14] J. N. Hooker and V. Vinay. Branching rules for satisfiability. *Journal of Automated Reasoning*, 15(3):359–383, 1995.

[15] Bernard Jurkowiak, Chu Min Li, and Gil Utard. A parallelization scheme based on work stealing for a class of sat solvers. *Journal of Automated Reasoning*, 34(1):73–101, 2005.

[16] Henry A. Kautz and Bart Selman. Planning as satisfiability. In *Proc. of the Tenth European Conference on Artificial Intelligence (ECAI'92)*, pages 359–363, 1992.

[17] W. Küchlin and C. Sinz. Proving consistency assertions for automotive product data management. *Journal of Automated Reasoning*, 24(1-2):145–163, February 2000.

[18] J. P. Marques-Silva and K. A. Sakallah. Grasp - a new search algorithm for satisfiability. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pages 220–227, 1996.

[19] J. P. Marques-Silva and K. A. Sakallah. Boolean satisfiability in electronic design automation. In *Proceedings of IEEE/ACM Design Automation Conference*, June 2000.

[20] F. Massacci and L. Marraro. Logical cryptanalysis as a SAT problem. *Journal of Automated Reasoning*, 24(1-2):165–203, February 2000.

[21] Carsten Sinz, Andreas Kaiser, and Wolfgang Küchlin. Formal methods for the validation of automotive product configuration data. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 17(1):75–97, January 2003. Special issue on configuration.

[22] H. Zhang, M. P. Bonacina, and J. Hsiang. PSATO: A distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation*, 21:543–560, 1996.

[23] L. Zhang and S. Malik. The quest for efficient boolean satisfiability solvers. In *Proc. of 8th International Conference on Computer Aided Deduction(CADE 2002)*, Copenhagen, Denmark, 2002.

Asynchronous iterative computations with Web information retrieval structures: The PageRank case

Giorgos Kollias[a], Efstratios Gallopoulos[a], Daniel B. Szyld[b]

[a]Computer Engineering and Informatics Department, University of Patras, GREECE

[b]Department of Mathematics, Temple University, Philadelphia, USA

1. Introduction

There are several ideas being used today for Web information retrieval, and specifically in Web search engines [20]. The PageRank algorithm [22] is one of those that introduce a content-neutral ranking function over Web pages. This ranking is applied to the set of pages returned by the Google search engine in response to posting a search query. PageRank is based in part on two simple common sense concepts: (i) A page is important if many important pages include links to it. (ii) A page containing many links has reduced impact on the importance of the pages it links to.

In this paper we focus on asynchronous iterative schemes [9,15] to compute PageRank over large sets of Web pages. The elimination of the synchronizing phases is expected to be advantageous on heterogeneous platforms. The motivation for a possible move to such large scale distributed platforms lies in the size of matrices representing Web structure. In orders of magnitude: 10^{10} pages with 10^{11} nonzero elements and 10^{12} bytes just to store a small percentage of the Web (the already crawled); distributed memory machines are necessary for such computations. The present research is part of our general objective, to explore the potential of asynchronous computational models as an underlying framework for very large scale computations over the Grid [14]. The area of "internet algorithmics" appears to offer many occasions for computations of unprecedent dimensionality that would be good candidates for this framework.

After giving a formulation of PageRank and its common interpretations in Section 2, we present its treatment under synchronous computational models. We next consider the asynchronous approach and comment on key aspects, specifically convergence, termination detection and implementation. In Section 5, we describe the experimental framework and present preliminary numerical experiments, while in Section 6 we draw our conclusions and discuss our future work on this topic. In this paper, as is common practice, we do not address the effects of finite precision arithmetic and roundoff error.

2. Formulation and Interpretations

In order to appreciate the PageRank computation, we present its standard formulation using the following set of four $n \times n$ matrices, where n is the number of pages being modeled.

An *adjacency matrix* A can be obtained through a web crawl or synthetically generated using statistical results, e.g., as in [10]. Thus, $A_{ij} = 1$ *iff* page i points to page j, and $A_{ij} = 0$ otherwise.

A *transition matrix* P has nonzero elements $P_{ij} = A_{ij}/\deg(i)$ when $\deg(i) \neq 0$, and zero otherwise (in which case page i is called a *dangling* page); here $\deg(i) = \sum_j A_{ij}$ is the outdegree of page i.

A *stochastic matrix* S is given by $S = P^T + w\, d^T$; $w = \frac{1}{n}e$, where e is the size n vector of all 1's, and d is the *dangling index vector* whose nonzero elements are $d_i = 1$ *iff* $\deg(i) = 0$.

The *Google matrix* G is $G = \alpha\, S + (1-\alpha)\, v\, e^T$. For a random web surfer about to visit his next page, the relaxation parameter α is the probability of choosing a link-accessible page. In choosing

otherwise, i.e., with probability $1-\alpha$, from the complete Web page set vector v contains respective conditional probabilities of such *teleportations*. Typically $v = w$ and $\alpha = 0.85$.

The PageRank vector x is the solution of the linear system

$$x = G\,x\,, \tag{1}$$

where the matrix G is an irreducible stochastic matrix, and thus its largest eigenvalue in magnitude is $\lambda_{max} = 1$ [26]. Thus, the PageRank vector x is the eigenvector corresponding to $\lambda_{max} = 1$, and when normalized, it is the reachability probability in a random walk on the Web, i.e., the invariant measure or stationary probability distribution of a Markov process modeled by the matrix G. It can also be computed as the solution to a system of linear equations. Using the fact that x is normalized to unity, i.e., $e^T\,x = 1$, equation (1) yields

$$(I-R)\,x = b \tag{2}$$

where $b = (1-\alpha)\,v$ and $R = \alpha\,S$ is the *relaxed stochastic matrix* [13].

3. Synchronous PageRank

To make the computation of x practical for the problem sizes we are considering, it is necessary to employ an iterative method, e.g., executing until convergence

$$x(t+1) \leftarrow f(x(t)) \tag{3}$$

with $t = 0,1,\ldots$ for a suitable operator, f and some initial vector $x(0)$. The vector $x(t)$ denotes the approximation to x obtained after t iterations. The above process needs to be mapped on a specific execution environment, corresponding to a computational model that typically preserves the semantics of the mathematical model in (3). The environment constitutes a virtual machine for the computation and is largely characterized by the types of units of execution (UE) (e.g., processes, threads) and communication mechanisms (e.g., shared memory, message passing) it readily supports, especially in hardware. Execution and communication entities are ultimately hosted by actual machines typically attached to nets (e.g., clusters) and internets (e.g., the Internet).

In the single UE case the aforementioned mapping on the execution environment is straightforward. For multiple UEs, however, this requires care: In the shared memory case, a semantics preserving mapping must involve synchronized access to shared memory cells between cooperating UEs, protected by locks, whereas in the message passing case, this synchronization is achieved through a barrier mechanism implemented atop collective blocking communication. For the PageRank computation, we can easily turn (1) into the following simple iteration:

$$x(t+1) = G\,x(t), \quad x(0) \text{ given.} \tag{4}$$

That is, $f(\cdot)$ amounts to a matrix-vector multiplication. This is the well-known power method for finding the eigenvector of G corresponding to the eigenvalue of largest magnitude [26], except that no per-step normalization needs to be performed. The normalization is not needed since a stochastic matrix such as G does not alter $\|x(t)\|_1$; and thus no danger of overflow or underflow is present here.

Single UE implementations of (4) with an emphasis on convergence acceleration, support for personalization through different teleportation vectors and utilization of naturally occurring block structure in the adjacency matrix A can be found in [17–19]. For multiple UEs, message passing computation of PageRank using the formulation (2) was presented in [16].

4. Asynchronous PageRank

Unfortunately, the necessary per-step synchronization of the synchronous algorithm described above grows into a significant overhead, especially as it is governed by the rate of the slowest UE and the costs of lock or barrier management. One radical transformation to harness this problem is to reduce the requirement for synchronization, e.g., by using non-blocking access to shared memory cells or network buffers. A central theme of our work is to investigate the effect of this transformation on the convergence, speed and overall effectiveness of the computations.

For an environment with p UEs, denote by $x_{\{i\}}$ the set of indices assigned to i^{th} UE during the iterative computation, T^i the set of times at which $x_{\{i\}}$ is updated (i.e., i^{th} UE finishes its computation) and $\tau^i_j(t)$ the time when the fragment $x_{\{j\}}$, which is available at time t in the i^{th} UE, was actually produced at its respective j^{th} UE. Then for $t \in T^i$, the i^{th} UE updates

$$x_{\{i\}}(t+1) \leftarrow f_i(x_{\{1\}}(\tau^i_1(t)),\ldots,x_{\{p\}}(\tau^i_p(t))), \tag{5}$$

while $x_{\{i\}}(t+1) = x_{\{i\}}(t)$ at other times. Delays due to omission of synchronization phases are expressed as differences $t - \tau^i_j(t) \geq 0$. The relation (5) is the asynchronous analog of (3) where f_i expresses the distributed operator component executing at the i^{th} UE. Obviously the form of f_i is independent of the asynchronism introduced. It thus follows that the normalization-free power method for PageRank computation at the i^{th} UE reads

$$x_{\{i\}}(t+1) = G_i\,[x^\top_{\{1\}}(\tau^i_1(t)),\ldots,x^T_{\{p\}}(\tau^i_p(t))]^\top \tag{6}$$

for $t \in T^i$, and $x_{\{i\}}(t+1) = x_{\{i\}}(t)$ at other times, where G_i is a set of rows of the Google matrix G indexed by $\{i\}$. Alternatively, while the synchronous, linear system equation approach would lead to an iterative scheme of the form $x(t+1) = R\,x(t) + b$ which can be seen to be identical to (4), its asynchronous formulation would lead to another, slightly different computational kernel, namely

$$x_{\{i\}}(t+1) = R_i\,[x^\top_{\{1\}}(\tau^i_1(t)),\ldots,x^T_{\{p\}}(\tau^i_p(t))]^\top + b_i \tag{7}$$

for $t \in T^i$, and $x_{\{i\}}(t+1) = x_{\{i\}}(t)$ at other times, at the i^{th} UE. Here R_i is a set of rows of the relaxed stochastic matrix R indexed by $\{i\}$, and b_i is the corresponding set of elements of vector b.

Also of interest are P2P computations of PageRank [12,23,25,29] These fall into the multiple UEs, message passing category and are asynchronous in nature. An important novelty in these studies is the dynamically generated link information through a notification protocol proposed to be integrated with the host Web servers.

The lack of synchronization annuls the semantics of the original mathematical algorithm. Therefore, it becomes necessary to discuss the convergence properties of the asynchronous scheme (5). We discuss this and related issues in the remainder of this section.

4.1. Convergence

Convergence of asynchronous iterative algorithms is usually established through constructing a sequence of nested boxed sets in the spirit of the following theorem [9]:

Theorem 1 *Let* $\{X(k)\} : \ldots \subset X(k+1) \subset X(k) \subset \ldots \subset X$, *with the following two conditions.*
Synchronous Convergence Condition: For all $k = 1,\ldots, x \in X(k)$, $f(x) \in X(k+1)$, *and for* $\{y^k\}, y^k \in X(k)$: *the limit points of* $\{y^k\}$ *are fixed points of* f.
Box Condition: For all $k = 1,\ldots, X(k) = X_1(k) \times \ldots \times X_p(k)$.
Then if $x(0) \in X(0)$, *the limit points of* $\{x(t)\}$ *are fixed points of* f, *where* $\{x(t)\}$ *are given by (5).*

Process (6) involves a nonnegative matrix of unit spectral radius; it is proved in [21] that the corresponding asynchronous iteration converges to the true solution within a multiplicative factor that can easily be factored out in the end by renormalization. A discussion on the misconception by some authors that for a nonnegative matrix B, spectral radius $\rho(B) < 1$ is a necessary condition for convergence of an asynchronous normalization-free power method can be found in [27]. On the other hand, process (7) involves a matrix R with $\rho(R) < 1$. Asynchronous iterations with such matrices are well known to converge to the true solution [9].

computing UE	**monitor UE**

if(checkConvergence())
  if(not converged)
  converged = true
  pc++
  if(pc = pcMax)
  send(CONVERGE, monitor)
  recv(STOP, monitor)
else
  if(converged)
  converged = false
  send(DIVERGE, monitor)
  pc = 0
	recv(CONVERGE | DIVERGE, all)
	if(checkConvergence())
	  if(not converged)
	  converged = true
	  pc++
	  if(pc = pcMax)
	  send(STOP, all)
	else
	  if(converged)
	  converged = false
	  pc = 0

Figure 1. `pc`: persistence counter, `pcMax`: its max value; reaching it triggers `CONVERGE/STOP` messages. They can have different values in monitor, computing UEs; all: all computing UEs

4.2. Termination Detection

The termination of asynchronous iterative algorithms is a non-trivial matter since local convergence at an UE does not automatically ensure global convergence. Even in the extreme case when all UEs have locally converged, one can devise scenarios where messages not yet delivered could destroy local convergence.

Both centralized and distributed protocols for termination detection can be found in the literature, [8,24]. In a centralized approach, a special UE acts as a monitor of the convergence process of other computing UEs; it keeps a log of the convergence status and issues `STOP` messages to all computing UEs when all of them have signaled their local convergence. In fact, computing UEs can issue either `CONVERGE` (when achieving local convergence) or `DIVERGE` (when exiting such a state) messages to the monitor UE. Distributed protocols for global convergence detection (see, e.g., [28]) are flexible but rather complex to implement. They typically assume a specific underlying communication topology. For example in [6] a leader election protocol is used, which in turn assumes a tree topology.

Our draft version of a practical centralized protocol, in part inspired by [7], is presented in Figure 1. It enforces *persistence* of convergence both at the computing UEs (for issuing a `CONVERGE` message) and at the monitor UE (for issuing a `STOP` message). Persistence is introduced to provide time for pending -and perhaps divergence causing- messages to be actually delivered.

4.3. Implementation

We focus on multiple UEs message passing environments, which is the case for our experiments. In that case, we need non-blocking communication primitives. These are actually implemented either by using multithreading (e.g., one thread per communication channel) or by multiplexing such channels and probing from within a single thread (through `select()` type mechanisms) for new data. Since multiple messages might have been received in the meantime, messages should be kept in queues organized under a common discipline.

5. Numerical Experiments

5.1. Application Structure

Our application consists of scripts steering `Java` classes. These scripts are written in `Jython` [2], which is an implementation of the `Python` [4] programming language in `Java` [1]. Such a mixed-language approach facilitates writing portable, interactive, easily extensible and flexible systems; after all, performance critical operations can always be isolated into compiled `.class` code.

Scripts build and use objects. `Configuration` objects can load/store parameters from/to configuration files - accessible from all other objects, partition and distribute matrix or vector data and optionally send code or launch processes over the cluster nodes. `Computation` objects perform computations and exchange information related to convergence status with `Monitor` objects implementing the termination detection protocol; cf. Figure 1. Communications are established through `Communication` objects which set up suitable communicators upon their instantiation; these communicators expose communication primitives to be invoked at each step.

We use multithreading in order to implement non-blocking communications. An asynchronous `send()` or `recv()` is just its blocking counterpart wrapped in a thread object and submitted to a thread pool endowed with a suitable task-handling strategy. Data are imported/exported through read/write channels with locks synchronizing those concurrently executing threads which happen to be managing messages with identical source and target IDs. Access to thread pool queues and pending communication-task-handles is provided so that a customized thread-management policy can be applied. At startup, a single file containing computation parameters should be available. This file is used by a `Configuration` object for the generation of node-specific configuration files and a script for distributing these files (optionally with other data or updated source code files) to the cluster nodes and initiating the computation. An option for automatic report generation is also provided.

5.2. Numerical Results

We used a Beowulf cluster of Pentium-class machines at 900 MHz, with 256 MB RAM each, running Linux, version 2.4 and connected to a 10 Mbps Ethernet LAN. We used Java 5.0, Jython 2.1 and *Matrix Toolkits for Java* [3] for composing our scripts and classes, all freely available on the Web. We report on some of the results of this ongoing work. The transition matrix used in the experiments is the Stanford-Web matrix [5], generated from an actual web-crawl. It contains connectivity info for $281,903$ pages ($2,312,497$ non-zero elements, 172 dangling nodes). We used the computational kernel (6) with a local convergence threshold of 10^{-6}. Note that in each case, blocks of consecutive $[n/p]$ rows were distributed among computing machines. Termination detection used `pcMax = 1` on both monitor and computing UEs. Configurations with 2, 4, and 6 machines were tested for both synchronous and asynchronous computations. Results in Table 1 are encouraging. On the other hand, it is fair to note that they correspond to reaching local convergence threshold. Assembling vector fragments resulting from asynchronous computations at monitor UE and then checking global

	Synchronous		**Asynchronous**		
procs	*iters*	*t* **(sec)**	$[iters_{min}, iters_{max}]$	$[t_{min}, t_{max}]$ **(sec)**	$\langle speedUp \rangle$
2	44	179.2	[68, 69]	[86.3, 94.5]	1.98
4	44	331.4	[82, 111]	[139.2, 153.1]	2.27
6	44	402.8	129, 148]	[141.7, 160.6]	2.66

Table 1
Numerical results: For the asynchronous case iteration ranges, computation time ranges are given ([max, min] values) since local convergence threshold is not 'simultaneously' reached at all nodes. A column with the average speedup offered by asynchronous computation over synchronous one is given (averages are over extreme values in the asynchronous case).

convergence reveals that a threshold of the order of 5×10^{-5} has actually been reached. Preliminary results of timing with respect to reaching a common global threshold (instead of a local one) reveals a modest speedup of asynchronous vs. synchronous computation in the $10-20\%$ range. Responsibility for the degradation of performance when increasing the number of UE's appears to lie with the overall large communication-to-computation ratio of the current algorithm. Observe, however, that what is important are not the accurate values of the PageRank vector components, but their relative ranking. Therefore, an issue in our present investigations is the effect of a more relaxed global threshold criterion on the computed page ranks.

Asynchronous iterative algorithms also seem to naturally adapt to heavy communication demands in a computation; current `send()/receive()` threads can block but computation thread is free to advance to next step iteration. On the contrary, in synchronous mode, no option exists except for blocking all threads (even the computation one), until data emitted from all nodes actually reach their destinations and synchronization completes, no matter whether the supporting network's characteristics suffice. In this case asynchronous computation can exhibit a low message import ratio (always with respect to iteration count which is obviously increased relative to synchronous setting); see Table 2.

	Sender				
Receiver	*id = 0*	*id = 1*	*id = 2*	*id = 3*	**Completed Imports (%)**
id = 0	109	46	23	26	29
id = 1	40	107	22	27	28
id = 2	35	37	111	66	41
id = 3	27	30	54	82	45

Table 2
Completed imports for the 4 computing UEs, asynchronous case. Rows contain the number of different vector fragments actually received during the computation from peers with respective IDs. Diagonal numerical entries contain the total number of locally computed and thus locally used vector fragments. *Completed Imports* column contains percentage averages of imports actually completed (should all be 100% for the synchronous case).

6. Conclusions and Future Work

The major performance bottleneck in our experiments to date is due to the large volume of data and the frequency that it is being produced. The latter is caused by the small computation time per-iteration (sparse matrix-vector multiplication). Note also that the communication pattern is an all-to-all scheme at each step; all these factors conspire to surpass the available network bandwidth and thus build memory consuming buffers of pending messages at the sending ends.

In the case of asynchronous iterations, data is being produced at a rate that is even higher than in the synchronous case, because part of the time gained from eliminating the synchronization phases is actually used for the production of extra messages; these (favorably) advance local iteration counters but they could also (unfortunately) overload the network; we guard against this misfortune by cancelling `send()/recv()` threads not having completed within a time window. The following is thus a hardly surprising conclusion from our experiments: Asynchronous iterative algorithms make up an alternative computation methodology in distributed environments. However this is not a black-box methodology and is most effectively utilized by iterative methods with heavy computational component and light communication. A frequent, all-to-all, fat message passing can saturate network infrastructure capacity, even in modest but dedicated cluster environments; heterogeneous environments like the Grid would be even more sensitive to such message passing scenarios. We would thus like to avoid the use of all-to-all communication schemes; after all the flexibility of asynchronous iterations gives us a choice on the targets of produced messages. Furthermore, it is advisable to employ an adaptive communication scheme; if message sending/receiving tasks fail to complete within a number of local iterations, reduce the rate of message exchanges with this not well 'responding' node.

In our ongoing work, we explore adaptive schemes for the asynchronous computation of PageRank. We also experiment with `select()` based implementations of asynchronism in order to amortize thread management costs. Since trees are naturally occurring internetwork topologies we also plan to study the performance of moving a clique-based (i.e., all-to-all) synchronous iterative method to an asynchronous, tree-based counterpart. We are also considering the use of suitable permutations (cf. [11]) as well as larger data sets.

Acknowledgments

The work of the first two authors was partially supported by a Hellenic Pythagoras-EPEAEK-II research grant. The work of the third author was partially supported by the US NSF grant CCF-0514489. The authors would also like to thank Professor G. Kallos for providing access to the computational facilities of the Physics Department, University of Athens.

References

[1] Java language website. `http://java.sun.com`.

[2] Jython website. `http://www.jython.org`.

[3] Matrix Toolkits for Java website. `http://www.math.uib.no/ bjornoh/mtj/`.

[4] Python language website. `http://www.python.org`.

[5] Stanford Web Matrix. `http://nlp.stanford.edu/ sdkamvar/data/stanford-web.tar.gz`.

[6] J.M. Bahi, S. Contassot-Vivier, R. Couturier, and F. Vernier. A Decentralized Convergence Detection Algorithm for Asynchronous Parallel Iterative Algorithms. *IEEE Trans. Parallel Distrib. Syst.*, 16:4–13, 2005.

[7] J.M. Bahi, S. Domas, and K. Mazouzi. Jace: A Java Environment for Distributed Asynchronous Iterative

Computations. In *EUROMICRO-PDP'04*, pages 350–357. IEEE, 2004.

[8] D.El Baz. A method of terminating asynchronous iterative algorithms on message passing systems. In *Parallel Algorithms and Applications*, 9:153–158, 1996.

[9] D.P. Bertsekas and J.N. Tsitsiklis. *Parallel and Distributed Computation*. Prentice Hall, Englewood Cliffs, NJ, 1989.

[10] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A.Tomkins, and J. Wiener. Graph structure in the web: experiments and models. In *9th Int'l. WWW Conf.*, 2000.

[11] H. Choi and D.B. Szyld. Application of threshold partitioning of sparse matrices to Markov chains. In *IPDS'96*, pages 158–165. IEEE, 1996.

[12] D. de Jager. PageRank: Three Distributed Algorithms. Master's thesis, Imperial College of Science, Technology and Medicine, London, Sept. 2004.

[13] G.M. Del Corso, A. Gulli, and F. Romani. Fast PageRank computation via a sparse linear system. In *Lecture Notes in Computer Science, Vol. 3243*, pages 118–130. 2004.

[14] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann - Elsevier, San Francisco, 2004.

[15] A. Frommer and D.B. Szyld. On asynchronous iterations. *J. Comput. Appl. Math.*, 123:201–216, 2000.

[16] D. Gleich, L. Zhukov, and P. Berkhin. Fast Parallel PageRank: A Linear System Approach. Technical report, Yahoo! Inc., 2004.

[17] T.H. Haveliwala, S.D. Kamvar, and G. Jeh. An Analytical Comparison of Approaches to Personalizing Pagerank. Technical report, Stanford Univ., July 2003.

[18] S.D. Kamvar, T.H. Haveliwala, C. D. Manning, and G.H. Golub. Exploiting the Block Structure of the Web for Computing PageRank. Technical report, Stanford Univ., March 2003.

[19] S.D. Kamvar, T.H. Haveliwala, C. D. Manning, and G.H. Golub. Extrapolation Methods for Accelerating PageRank Computations. In *Proc. 12th Int'l. WWW Conf.*, May 2003.

[20] A.N. Langville and C.D. Meyer. A Survey of Eigenvector Methods for Web Information Retrieval. *SIAM Rev.*, 47:135–161, 2005.

[21] B. Lubachevsky and D. Mitra. A Chaotic, Asynhronous Algorithm for Computing the Fixed Point of a Nonnegative Matrix of Unit Spectral Radius. *J. ACM*, 33:130–150, Jan. 1986.

[22] L. Page, S. Brin, R. Montwani, and T. Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical report, Stanford Univ., 1998.

[23] K. Sankaralingam, S. Sethumadhavan, and J. C. Browne. Distributed Pagerank for P2P Systems. In *12th Int'l. Symposium on High Performance Distributed Computing*, 2003.

[24] S.A. Savari and D.P. Bertsekas. Finite termination of asynchronous iterative algorithms. *Parallel Computing*, 22(1):39–56, 1996.

[25] S.-M. Shi, J. Yu, G. Yang, and D. Wang. Distributed Page Ranking in Structured P2P Networks. In *ICPP'03*. IEEE, 2003.

[26] W.J. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton Univ. Press, 1994.

[27] D.B. Szyld. The mystery of asynchronous iterations convergence when the spectral radius is one. Technical Report 98-102, Department of Mathematics, Temple Univ., Philadelphia, Oct. 1998.

[28] A.S. Tanenbaum and M. van Steen. *Distributed Systems, Principles and Paradigms*. Prentice-Hall, Upper Saddle River, NJ, 2002.

[29] L. Tsimonou. Distributed PageRank: Comparisons between a Simulation and a Peer-to-Peer Implementation of the Algorithm. Master's thesis, Imperial College of Science, Technology and Medicine, London, Sept. 2004.

Solving Real Life Applications With High Accuracy

Carlos Amaral Hölbig[a], Paulo Sérgio Morandi Júnior[b], Dalcidio Moraes Claudio[c], Tiarajú Asmuz Diverio[b]

[a]Universidade de Passo Fundo – ICEG – Passo Fundo – Brazil

[b]Universidade Federal do Rio Grande do Sul – II and PPGC – Porto Alegre – Brazil

[c]Pontifícia Universidade Católica do Rio Grande do Sul – Porto Alegre – Brazil

1. Introduction

In this paper we discuss the implementation of methods to solve linear systems, with high accuracy, used in real life applications. It is important because many different numerical algorithms of these applications contain the solution of linear system (1) as a subproblem. Because of these aspects, this work aims the development of solvers to linear systems (for dense and sparse matrices) with high accuracy on cluster computers using C–XSC library. The methods implemented are used in real life applications like hydrodynamic, agriculture and power electric systems.

$$Ax = b \tag{1}$$

In our research some programs were developed in C–XSC with the high accuracy characteristic, where the results are obtained with a good quality. The C-XSC library is a (free) C++ class library for scientific computing for the development of numerical algorithms delivering highly accurate and automatically verified results by use of the interval arithmetic (see details about this library in [1] and [2]). It provides a large number of predefined numerical data types and operators. These types are implemented as C++ classes. Thus, C-XSC allows high-level programming of numerical applications in C++. Actually, our software run on clusters at UFRGS (Brazil), UPF (Brazil) and Wuppertal (Germany).

During the development of this work was necessary to available a high performance (cluster computers) and high accuracy (by use of C–XSC library) environment. Because of this, our work was divided in 4 (four) steps:

- Integration between the libraries C–XSC and MPI on clusters (section 2);
- Development of solvers for linear systems (section 3);
- Implementation of basic tests in the environment (section 4);
- Solve real life applications with high accuracy (section 5).

Because of these aspects, this work aims the development of solvers with high accuracy for linear systems of equations and the adaptation of the algorithms implemented to cluster computers using C–XSC library. This library is available for download in *www.math.uni-wuppertal.de/wrswt/index_en.html*. Our solvers work with dense and sparse (in special banded matrices) linear equation systems. Nowadays, the solver for dense matrices works with all four basic numerical C–XSC data types: *real, interval, complex*, and *complex interval* and the solver for sparse matrices works with *real* and *interval* data types. All our programs are freeware [3].

2. Integration between C–XSC and MPI Libraries

As part of our research, we done the integration between C–XSC and MPI libraries on cluster computers. This step was necessary and essential for the adaptation of our solvers to high performance environments. This integration was developed using, initially, algorithms for matrix multiplication in parallel environments of cluster computers. We done some comparations about the time related to the computational gain using parallelization, the parallel program performance depending on the matrix order and the parallel program performance using a larger number of nodes. We also studied some other information like the memory requirement in each method to verify the performance relation with the execution time and memory. The initial tests of integration has developed on labtec cluster at II-UFRGS (cluster with 20 Dual Pentium III 1.1 GHz (40 nodes), 1 GB memory RAM, HD SCSI 18 GB and Gigabit Ethernet; cluster server (front-end) with Dual Pentium IV Xeon 1.8 GHz, 1 GB memory RAM, HD SCSI 36 GB and Gigabit Ethernet). We want to join the high accuracy given by C-XSC with the computational gain provided by parallelization [4].

This parallelization was developed with the tasks division among various nodes on cluster. These nodes execute the same kind of tasks and the communication between the nodes and between the nodes and the server uses message passing protocol. About the C–XSC programs executed on cluster, some changes were made in the programs for their correct use in this environment, mainly about how to manipulate *dotprecisions* variables (high accuracy variables of C–XSC) [5].

3. Solvers for Linear Systems

The algorithms implemented in our solvers were described in [6] and can be applied to any system of linear equations which can be stored in the floating point system on the computer. They will, in general, succeed in finding and enclosing a solution or, if they do not succeed, will tell the user so. In the latter case, the user will know that the problem is very ill conditioned or that the matrix A is singular. In the implementation in C–XSC, there is a chance that if the input data contains large numbers or if the inverse of A or the solution itself contain large numbers, an overflow may occur, in which case the algorithms may crash. In practical applications, this has never been observed, however. This could also be avoided by including the floating point exception handling which C–XSC offers for IEEE floating point arithmetic [7].

For this work we implemented interval algorithms for solution of linear systems of equations with dense and sparse matrices [5,8,9]. There are numerous methods and algorithms computing approximations to the solution x in floating-point arithmetic. However, usually it is not clear how good these approximations are, or if there exists a unique solution at all. In general, it is not possible to answer these questions with mathematical rigour if only floating-point approximations are used. These problems become especially difficult if the matrix A is ill conditioned. We present some algorithms which answer the questions about existence and accuracy automatically once their execution is completed successfully. Even very ill conditioned problems can be solved with these algorithms. Most of the algorithms presented here can be found in [10].

3.1. Solver for Dense Linear Systems

The C-XSC programs implemented in our solver for dense linear systems were written for the case of *real* input data (i.e. A is of type *rmatrix* and b is of type *rvector*) and for the case of the data types *interval, complex*, and *complex interval*. The changes made for the use of these other types are mainly changes of the data type of certain variables and functions in the program. Our C–XSC program *verifies the existence* of a solution and *computes an enclosure* for each of the following

types of problems:

(s) compute an enclosure for the solution of system (1) for a *square* $n \times n$ matrix A.

(o) compute an enclosure for the solution of system (1) in the *over-determined* case, i.e. for an $m \times n$ matrix A where $m > n$.

(u) compute an enclosure for the solution of system (1) in the *under-determined* case, i.e. for an $m \times n$ matrix A where $m < n$.

(S) compute an enclosure of the *inverse* A^{-1} of A.

(O) compute an enclosure of the *pseudo inverse* A^{+} of A in the *over-determined* case, i.e. for an $m \times n$ matrix A where $m > n$.

(U) compute an enclosure of the *pseudo inverse* A^{+} of A in the *under-determined* case, i.e. for an $m \times n$ matrix A where $m < n$.

This solver has two modules: the module `lss_aprx` contains the function `MINV` which computes an approximate inverse of the input matrix A of type *rmatrix* using the Gauss-Jordan algorithm (see i.e. [11]), when A is a square matrix. In the over- or under-determined case we use the Moore-Penrose pseudo inverse A^{+} of A (if A has full rank). The second module `lss` contains the functions which solve the dense linear system. This system may be square and non square ($m \times n$). In the over-determined case ($m > n$) a vector $x \in \mathbb{R}^n$ is sought whose residuum $b - Ax$ has minimal Euclidian norm whereas in the under-determined case ($n < m$) a solution $x \in \mathbb{R}^n$ is sought which has minimal norm.

3.2. Solver for Sparse Linear Systems

For the solution of a sparse linear system we present an implementation of an algorithm to compute efficiently componentwise good enclosures. Our implementation works with point as well as *interval* data (data afflicted with tolerances). We assume linear systems whose coefficient matrix has a banded structure. In this case the well known general algorithm (using the Krawczyk operator) to solve systems with dense matrices is not efficient. Since the approximate inverse R of a banded matrix A is in general a full matrix, a lot of additional storage would be required, especially if the bandwidth of A is small compared with its dimension. So a special algorithm is used to reduce the amount of storage and runtime. This method is based on the fact that matrices with banded structure are closely related to difference equations. For the banded system, we apply a LU-decomposition without pivoting (to avoid fill in) to the coefficient matrix A and derive an interval iteration similar to the well known interval iteration used in case of dense matrices. Here, however, we do not use a full approximate inverse R, but rather the interval iteration will be performed by solving two systems with banded triangular matrices L and U. The banded triangular systems are solved with the special method for difference equations described in [6]. In case of point matrices the method is designed to give almost sharp enclosures for all components (large or small in modulus) of the solution vector. A different approach to compute an enclosure for the solution vector of a large linear systems with banded or arbitrary sparse coefficient matrix (which gives enclosures with respect to the infinity norm $||\ \ ||_\infty$ only) is described in [10].

In addition to the implementation of the solution method in C–XSC, the program includes a small demonstration part (a driver) which can be used to solve some simple systems. First the program reads the number of lower and upper bands and then one value for each of the bands, i.e. initially a

Toeplitz matrix is generated. In the next step, however, any number of elements of the matrix can be changed, such that arbitrary banded matrices can be entered. To change the element $a_{i,j}$, only i, j and the new value for this element must be entered. Changing of elements is finished by entering zeros for i and j. Next the right hand side must be entered. There are several choices of predefined solutions, such that the right hand side b will be determined from this given solution. Alternatively b can be set to a constant value in all components or all components can be entered successively. In any case, the values of the components of b may be changed again similarly as for the matrix. When no changes are done anymore, the solution algorithm starts. The banded solver is called and the solution and error statistics are printed. In this way it is quite easy to explore the our C–XSC solver.

4. Basic Tests and Results

Measures and tests were made to compare the routines execution time in C/C++ language, C/C++ using MPI library, C/C++ using C–XSC library and C/C++ using C–XSC and MPI libraries. Our first tests were made to compare the routines execution time, the accuracy of results and to validate the environment developed.

4.1. Scalar product

We done tests comparing the accuracy and execution time between programs in C/C++ language (scalar product) and C/C++ using C–XSC library (optimal scalar product - with high accuracy). In the tables 1, 2 and 3 we showed the results of test with scalar product between two vectors a and b. The values of this vectors are: $a = [10^{50}, 1.25, 10^{50}, 1.1, \ldots, 10^{50}, 1.25, 10^{50}, 1.1]$ and $b = [1, 1, -1, -1, \ldots, 1, 1, -1, -1]$ with size of this vectors (n) equal to $30000, 90000$ and 180000.

Table 1
Parallel Scalar Product $a \times b$ (using 4 nodes of cluster labtec – time in seconds)

Order (n)	Program in C/C++	Program in C–XSC
30000	0.031205	0.041015
90000	0.096727	0.113847
180000	0.188670	0.225083

Table 2
Results of Scalar Product $a \times b$ in C/C++

Order (n)	Results in C/C++	Exact Results
30000	$-3.300000000000000266453525910037569701671 6$	1125
90000	$-3.300000000000000266453525910037569701671 6$	3375
180000	$-3.300000000000000266453525910037569701671 6$	6750

Table 3
Results of Scalar Product $a \times b$ in C–XSC

Order (n)	Results in C–XSC	Exact Results
30000	1124.99999999999931787897367030382156372070 3	1125
90000	3374.99999999999818101059645414352416992187 5	3375
180000	6749.99999999999636202119290828704833984375 0	6750

4.2. Matrix multiplications

We done tests with sequential and parallel programs in C/C++ language, C/C++ using MPI library, C/C++ using C–XSC library and C/C++ using C–XSC and MPI libraries. The results about the performance are showed in the table 4.

Table 4
Parallel Matrix Multiplication (using 8 nodes of cluster labtec – time in seconds)

Program/Order	512×512	1024×1024	2048×2048
C/C++ with MPI	4.603	34.131	265.284
C–XSC with MPI	115.523	916.233	7329.415

4.3. Solvers to linear systems with dense and banded matrices

To validate our solvers on clusters we will describe some tests about the quality of the results of ours solvers. How a first example, a very well known set of ill conditioned test matrices for linear system solvers are the $n \times n$ Hilbert matrices H_n with entries $(H_n)_{i,j} := \frac{1}{i+j-1}$. As a test problem, we report the results of our program for the linear systems $H_n x = e_1$, where e_1 is the first canonical unit vector. Thus the solution x is the first column of the inverse H_n^{-1} of the Hilbert matrix H_n. Since the elements of these matrices are rational numbers which can not be stored exactly in floating point, we do not solve the given problems directly but rather we multiply the system by the least common multiple lcm_n of all denominators in H_n. Then the matrices will have integer entries which makes the problem exactly storable in IEEE floating point arithmetic. For the system $(lcm_{10} H_{10})x = (lcm_{10} e_1)$, the program computes the enclosures (here an obvious short notation for intervals is used) showed in (2), which is an extremely accurate enclosure for the exact solution (the exact solution components are the integers within the computed intervals).

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \end{pmatrix} = \begin{pmatrix} 1.000000000000000E+002, & 1.000000000000000E+002 \\ -\ 4.950000000000000E+003, & -\ 4.950000000000000E+003 \\ 7.920000000000000E+004, & 7.920000000000000E+004 \\ -\ 6.006000000000000E+005, & -\ 6.006000000000000E+005 \\ 2.522520000000000E+006, & 2.522520000000000E+006 \\ -\ 6.306300000000000E+006, & -\ 6.306300000000000E+006 \\ 9.609600000000000E+006, & 9.609600000000000E+006 \\ -\ 8.751600000000000E+006, & -\ 8.751600000000000E+006 \\ 4.375800000000000E+006, & 4.375800000000000E+006 \\ -\ 9.237800000000000E+005, & -\ 9.237800000000000E+005 \end{pmatrix} \tag{2}$$

As an other example, we compute an enclosure for a very large system. We take a symmetric Toeplitz matrix with five bands having the values $1, 2, 4, 2, 1$ and on the right hand side we set all components of b equal to 1. Then the program produces the following output for a system of size $n = 200000$ (only the first ten and last ten solution components are printed):

```
Dimension  n = 200000
Bandwidths l,k : 2 2
A = 1 2 4 2 1
change elements ? (y/n) n
```

```
b = =1
change elements ? (y/n) n

x =
  1: [  1.860146067479180E-001,  1.860146067479181E-001 ]
  2: [  9.037859550210300E-002,  9.037859550210302E-002 ]
  3: [  7.518438200412189E-002,  7.518438200412191E-002 ]
  4: [  1.160876404875081E-001,  1.160876404875082E-001 ]
  5: [  1.003153932563721E-001,  1.003153932563722E-001 ]
  6: [  9.427129202687645E-002,  9.427129202687647E-002 ]
  7: [  1.028361799416204E-001,  1.028361799416205E-001 ]
  8: [  1.005240450090008E-001,  1.005240450090009E-001 ]
  9: [  9.874921290539136E-002,  9.874921290539138E-002 ]
 10: [  1.004617422430963E-001,  1.004617422430964E-001 ]

199990: [  1.001953939326196E-001,  1.001953939326197E-001 ]
199991: [  1.004617422430963E-001,  1.004617422430964E-001 ]
199992: [  9.874921290539136E-002,  9.874921290539138E-002 ]
199993: [  1.005240450090008E-001,  1.005240450090009E-001 ]
199994: [  1.028361799416204E-001,  1.028361799416205E-001 ]
199995: [  9.427129202687645E-002,  9.427129202687647E-002 ]
199996: [  1.003153932563721E-001,  1.003153932563722E-001 ]
199997: [  1.160876404875081E-001,  1.160876404875082E-001 ]
199998: [  7.518438200412189E-002,  7.518438200412191E-002 ]
199999: [  9.037859550210300E-002,  9.037859550210302E-002 ]
200000: [  1.860146067479180E-001,  1.860146067479181E-001 ]

max. rel. error =  1.845833860422451E-016  at i = 3
max. abs. error =  2.775557561562891E-017  at i = 1
min. abs. x[3] = [7.518438200412189E-002, 7.518438200412191E-002]
max. abs. x[1] = [1.860146067479180E-001, 1.860146067479181E-001]
```

Our last example is about the matrix inversion (test about accuracy).

In this example $A = \begin{pmatrix} 1 & 1 & 0 \\ 0 & \epsilon & 0 \\ 0 & 0 & \epsilon^2 \end{pmatrix}$ and $A^{-1} = \begin{pmatrix} 1 & -\frac{1}{\epsilon} & 0 \\ 0 & \frac{1}{\epsilon} & 0 \\ 0 & 0 & \frac{1}{\epsilon^2} \end{pmatrix}$.

With $\epsilon = 1.084202172485504E-019$, our program in C–XSC obtained:

$$A = \begin{pmatrix} 1.0E+000, & 1.000000000000000E+000, & 0.000000000000000E+000 \\ 0.0E+000, & 1.084202172485504E-019, & 0.000000000000000E+000 \\ 0.0E+000, & 0.000000000000000E+000, & 1.175494350822288E-038 \end{pmatrix}$$

$$A^{-1} = \begin{pmatrix} 1.0E+000, & 1.000000000000000E+000, & 0.000000000000000E+000 \\ 0.0E+000, & 7.136238463529799E+044, & 0.000000000000000E+000 \\ 0.0E+000, & 0.000000000000000E+000, & 8.507059173023462E+037 \end{pmatrix}.$$

5. Solving Real Life Applications With High Accuracy

In the original solvers we included three new methods to solve linear systems: Conjugate Gradient, Householder and Givens methods. All these methods have versions with and without the high accuracy characteristic. We use this methods in real life applications like hydrodynamic (parallel computational model with local refinement and dynamic load balancing for the simulation of substances transportation and hydrodynamic – [12]), agriculture (optimization of the air distribution in grain storehouse with aeration of the mass of grains – [13]) and power electric systems [14]. This methods and applications are research topics of ours research groups in Brazil [9].

In our research we are implementing other methods to solve linear systems. We are implementing, initially, the sequential versions of Gauss-Seidel, Gauss-Jacobi, Gauss Elimination and LU Decomposition methods [15–17]. Nowadays, we are implementing the parallel versions of these methods with and without high accuracy (these programs are been implemented only with the high accuracy characteristic, we are not using interval arithmetic).

6. Conclusions

In our research some programs were developed in C–XSC with the validated numeric paradigm, where the results are obtained with a good quality. The main contributions of our work are:

- integration between the libraries C–XSC and MPI;
- effective use of library C-XSC on cluster computers;
- resolution of linear systems with high accuracy.

In our work we provide the development of selfverifying solvers for linear systems of equations with dense and sparse matrices and the integration between C–XSC and MPI libraries on cluster computers. This integration was not trivial because was necessary to send correctly the special high accuracy variables (*dotprecision*) of C–XSC to the cluster processors using the library MPI without lost of the high accuracy characteristic. Our software run on clusters at UFRGS, UPF and Wuppertal and the integration between C-XSC and MPI was done correctly. Our tests with matrix multiplication, scalar product and methods to solve linear system of equations show that the C-XSC library needs to be optimized to be efficient in a High Performance Environment. This optimization is other research topic of Brazilian groups.

Nowadays we are working in the implementation of parallel versions of methods to solve linear systems (without and with high accuracy). These methods are used in real life applications like hydrodynamic (parallel computational model with local refinement and dynamic load balancing for the simulation of substances transportation and hydrodynamic), agriculture (optimization of the air distribution in grain storehouse with aeration of the mass of grains) and power electric systems.

Acknowledgement

This work is supported in part by CAPES and FAPERGS (Brazil) and DAAD (Germany). It is part of a international cooperation project between Brazilian universities (Universidade Federal do Rio Grande do Sul, Pontifícia Universidade Católica do Rio Grande do Sul and Universidade de Passo Fundo) and German universities (Universität Wuppertal and Universität Karlsruhe).

References

[1] Hammer, R., Hocks, M., Kulisch, U., Ratz, D.: *C-XSC Toolbox for Verified Computing I: basic numerical problems*. Springer-Verlag, Berlin/Heidelberg/New York, 1995.

[2] Hofschuster, W., Krämer, W., Wedner, S., Wiethoff, A.: *C-XSC 2.0: A C++ Class Library for Extended Scientific Computing*. Universität Wuppertal, Preprint BUGHW - WRSWT 2001/1 (2001).

[3] Hölbig, C.A., Krämer, W.: *Selfverifing Solvers for Dense Systems of Linear Equations Realized in C-XSC*. Universität Wuppertal, Preprint BUGHW - WRSWT 2003/1, Wuppertal, 2003.

[4] Hölbig, C.A., Diverio, T.A., Claudio, D.M., Krämer, W., Bohlender, G.: Automatic Result Verification in the Environment of High Performance Computing In: IMACS/GAMM International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics, 2002, Paris. Extended abstracts, pg. 54-55 (2002).

[5] Hölbig, C.A., Morandi Júnior, P.S., Alcalde, B.F.K., Diverio, T.A., Claudio, D.M.: Solving Linear Systems on Cluster Computers with High Accuracy. In: VII WORKSHOP ON STATE-OF-THE-ART IN SCIENTIFIC COMPUTING, 2004, Copenhagen. Book of Abstracts, pg. 28–29. Copenhagen, 2004.

[6] Krämer, W., Kulisch, U., Lohner, R.: *Numerical Toolbox for Verified Computing II - Advanced Numerical Problems*. University of Karlsruhe (1994), see *http://www.uni-karlsruhe.de/~Rudolf.Lohner/papers/tb2.ps.gz*.

[7] American National Standards Institute / Institute of Electrical and Electronics Engineers: *A Standard for Binary Floating-Point Arithmetic*. ANSI/IEEE Std. 754-1985, New York, 1985.

[8] Hölbig, C.A., Krämer, W., Diverio, T.A.: *An Accurate and Efficient Selfverifying Solver for Systems with Banded Coefficient Matrix*. In [19], pp. 283–290, 2004.

[9] Hölbig, C.A., Kolberg, M.L., Morandi Júnior, P.S., Alcalde, B.F.K., Diverio, T.A., Claudio, D.M.: Solvers with High Accuracy to Linear Systems on Clusters. In: XI INTERNATIONAL CONGRESS ON COMPUTATIONAL AND APPLIED MATHEMATICS, 2004, Leuven. Abstracts of Talks, pg. 70. Leuven, 2004.

[10] Rump, S. M.: *Validated Solution of Large Linear Systems*. In [18], pp 191–212, 1993.

[11] Stoer, J., Bulirsch, R.: *Introduction to Numerical Analysis*. Springer-Verlag, New York, 1980.

[12] Rizzi, R.L., Dorneles, R.V., Piccinin Júnior, D., Martinotto, A.L., Hölbig, C.A., Navaux, P.O.A., Diverio, T.A.: *Parallelization of Krylovs Subspace Methods in Multiprocessor PC Cluster*. In [19], pp. 543–550, 2004.

[13] Khatchatourian, O., Savicki, L.D.: Optimization of the air distribution in Grain Storehouse with Aeration in No-Uniform Conditions of the Mass of Grains. In: 16th Brazilian Congress of Mechanical Engineering, 2001, Uberlândia. Proceedings, pp. 73–82, Uberlândia, 2001.

[14] Barboza, L.V., Dimuro, G.P., Reiser, R.H.S.: Towards Interval Analysis of the Load Uncertainty in Power Electric Systems. In: 8th International Conference on Probability Methods Applied to Power Systems, 2004, Washington. Proceedings, pp 1–6, Washington, 2004.

[15] Dongarra, J., Foster, I., Fox, G., Gropp, W., Kennedy, K., Torczon, L., White, A.: *Sourcebook of Parallel Computing*. Morgan Kaufmann Publishers, San Francisco. 2003.

[16] Karniadakis, G.E., Kirby II, R.M.: *Parallel Scientific Computing in C++ and MPI: A Seamless Approach to Parallel Algorithms and Their Implementation*. Cambridge University Press, Cambridge. 2003.

[17] Yousef Saad: *Iterative Methods for Sparse Linear Systems*. SIAM, Philadelphia. 2003.

[18] Albrecht, R., Alefeld, G., Stetter, H. J. (Eds.): *Validation Numerics – Theory and Applications*. Computing Supplementum **9**, Springer-Verlag (1993).

[19] Joubert, G.R., Nagel, W.E., Peters, F.J., Walter, W.V. (Org.): Parallel Computing: Software Technology, Algorithms, Architectures, and Applications. Elsevier Science Publishers, Londres, 2004.

Improving ease of use in BLACS and PBLAS with Python

Tony Drummond[a] , Vicente Galiano[b] , Violeta Migallón[c] , José Penadés[c]

[a]Lawrence Berkeley National Laboratory, One Cyclotron Road, Berkeley CA 94703, USA, LADrummond@lbl.gov

[b]Departamento de Física y Arquitectura de Computadores, Universidad Miguel Hernández, 03202 Elche, Alicante, Spain, vgaliano@umh.es

[c]Departamento de Ciencia de la Computación e Inteligencia Artificial, Universidad de Alicante, 03071 Alicante, Spain, {violeta, jpenades}@dccia.ua.es

Many computational applications rely heavily in high performing numerical linear algebra operations. A good number of these applications are data and computation intensive that need to run in high performance computing environments. Researchers and engineers behind these applications have to spend a considerable amount of time efficiently developing and running codes in these environments. Developers would rather devote this time at using their computational applications. To alleviate this, we promote the reuse of robust software libraries like the ones in the ACTS Collection, and here we present our work in a subset of the high-level language interfaces, PyACTS, that help users prototype their codes using these libraries. Lastly, we compare traditional programming practices to our proposed approach. We illustrate some examples of these interfaces and their performance, we also evaluate not only their performance but also how user friendlier they are compared to the original calls.

1. Introduction

In many scientific and engineering areas there is a need to solve computational problems inside simulations to study a particular physical phenomena. Researchers and engineers behind these applications have to spend a considerable amount of time efficiently developing and running codes in high performance computing environments. Developers would rather devote this time at using their computational applications and analyzing their output.

To alleviate this, we promote the reuse of robust software libraries like the ones in the ACTS Collection [8]. ScaLAPACK [2], is one of the tools in the ACTS Collection, and has widely been used in many high performance scientific applications [7]. ScaLAPACK internally implements parallelism and scalability with the use of BLACS [6] and PBLAS [11]. In many instances, scientists and engineers find difficulty understanding the interfaces to libraries like BLACS and PBLAS because they carry arguments that are related to the parallel environment and performance in addition to arguments related to the problem at hand.

We work in the development of a collection of Python interfaces to the ACTS tools, PyACTS, and here we present the work done to the BLACS and PBLAS libraries. These two libraries involve complex distributed data structures that our proposed interfaces, PyBLACS and PyPBLAS, hide from the final user, making them easier to use.

To create these interfaces, we have chosen the Python language due to several reasons that we explain in Section 2. In Section 3, we briefly introduce the PyACTS project and the core work presented here. In Sections 4 and 5, we present the PyBLACS and PyPBLAS libraries, respectively. We compare the performance of these libraries against the performance of their native versions.

2. Python and PyMPI

Python [14] is an interpreted, interactive, object-oriented programming language. Python combines remarkable power with very clear syntax. It has modules, classes, exceptions, very high level dynamic data types, and dynamic typing. There are interfaces to many system calls and libraries, as well as to various windowing managing systems. New built-in modules are easily written in C or C++.

Nowadays, scripting languages, and particularly Python language, have gained popularity inside the computational sciences community (see e.g., [12], [13]), in part because of the complexity of codes, limited computational resources, large volumes of data produced, setup of large simulation runs, and the integration of the fairly large individual codes. As computational sciences continue to generate advancements in sciences and engineering, we will continue to witness an increase in the complexity of the computational environments, software tools and applications.

In the original design, Python was not targeted for parallel programming although now it has a thread model. In order to be able to work in a parallel environment, we have evaluated two different implementations of the thread model: Scientific Python [9] and PyMPI [10]. Fundamentally, these tools are Python extension sets designed to provide parallel operations for Python on a distributed and parallel machine, using the standard Message Passing Interface (MPI) [5]. We have tested the libraries developed in this work with both interpreters and we have obtained similar results. The interpreter used in this work is PyMPI. In the rest of this section, we explain briefly the use of this interpreter.

PyMPI works by building an alternate startup executable for Python, and with this use all the installed base of Python code modules, which in turn enables PyMPI to use the same Python modules and rich functionality. By default, PyMPI starts up in MIMD mode by initiating multiple Python interpreters, each one with access to world communicators and with a unique rank corresponding to a process ID. That is, all processes are running the same script but they execute different instructions according to the value of the process ID. The execution is asynchronous unless synchronization operations are used. PyMPI can be used in both batch and interactive mode. In the batch mode, a Python script is needed as the input file on the command line (e.g., `mpirun -np 3 pyMPI script.py`). If starting up PyMPI in the interactive mode, we just fire it up in the same way as executing a parallel job and we get what looks like the standard Python prompt (`>>>`), as shown in Figure 1. As mentioned earlier, the processes are running asynchronously such that the values printed out may appear in any order.

```
% mpirun -np 3 pyMPI
>>> import mpi
>>> print 'Hello world',mpi.rank
Hello world 0
Hello world 2
Hello world 1
>>>
```

Figure 1. Running PyMPI interactively.

3. The PyACTS project

The ACTS Collection [8] is a set of software tools that help programmers to write high performance scientific codes. It differs from other "software tool" projects in that it focuses primarily on software used inside an application, instead of on software used to develop an application. ACTS is an umbrella project that has brought the tools together and is funding developers to provide interoperability. ACTS tools are mostly libraries (some are C libraries, some C++ class libraries, and some are Fortran libraries). They are primarily designed to run on distributed memory computers. Portability and performance were both considerations in their design.

We work in the development of PyACTS as a set of Python based modules that provide a high level user interface to functionality available in the ACTS Collection. With PyACTS, we will provide an interoperable environment, where different libraries can be used interchangeably. For this purpose, we define a new class object in Python: `PyACTS Array`. An instance of a `PyACTS Array` contains the following properties:

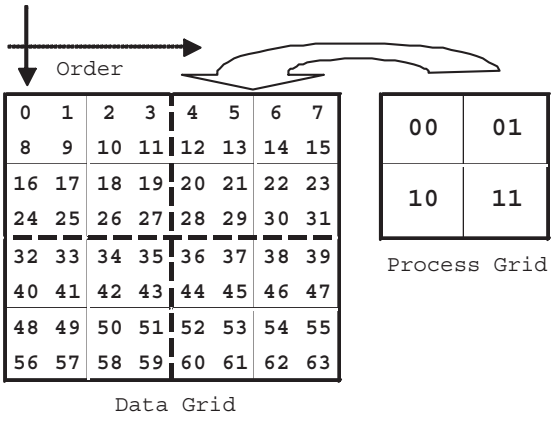
- **lib**: An integer identifies which ACTS library is used. This is useful for embedding the data types and data conversion mechanisms between different ACTS libraries.
- **desc**: Corresponds to the descriptor array of the distributed data in the selected library. Usually, libraries use descriptors to hold the global attributes.
- **data**: This property contains a Numeric Python Array (most frequently called `Numpy`) [1].

In order to provide a user friendly interface and offer a variety of levels of computational services, we must provide several groups of routines. These groups are divided as follows: *Basic Services* (a set of PyACTS routines that allows the creation, destruction, duplication, update and query of information), *I/O Services* (this group implements the PyACTS I/O routines), *Verification and Validation* (in this group, we implement routines to validate arguments and data used in PyACTS and its routines), *Errors and Exceptions* (this group implements the handling of computational or semantic errors detected during the execution) and *Data Conversion* (this group implements interoperability between different data structures used inside the different ACTS libraries).

convert.py

```
import PyACTS
import Numeric
PyACTS.gridinit(nb=2)
n,ACTS_lib=8,1                          # ScaLAPACK library
if PyACTS.iread==1:
   a=Numeric.reshape(range(n*n),[n,n])
else:
   a=None
a=PyACTS.Num2PyACTS(a,ACTS_lib)         # Convert Numpy to PyACTS
print "PyACTS Array Properties in [",
     PyACTS.myrow,",",PyACTS.mycol,"]"
print "       lib=",a.lib,";desc=",a.desc
print "       data=",a.data
PyACTS.gridexit()
```

Figure 2. Converting from Numpy to PyACTS.



(a) Data distribution

```
PyACTS Array Properties in [0,0]
    lib=1;desc=[1 0 8 8 2 2 0 0 4]
     data=[0  8 32 40  1 9 33 41
           4 12 36 44  5 13 37 45]
PyACTS Array Properties in [1,0]
    lib=1;desc=[1 0 8 8 2 2 0 0 4]
     data=[16 24 48 56 17 25 49 57
           20 28 52 60 21 29 53 61]
PyACTS Array Properties in [1,1]
    lib=1;desc=[1 0 8 8 2 2 0 0 4]
     data=[18 26 50 58 19 27 51 59
           22 30 54 62 23 31 55 63]
PyACTS Array Properties in [0,1]
    lib=1;desc=[1 0 8 8 2 2 0 0 4]
     data=[2 10 34 42  3 11 35 43
           6 14 38 46  7 15 39 47]
```

(b) Output

Figure 3. Results of executing convert.py.

In Figure 2, we show an example of converting data from Numpy to PyACTS. In this example we execute the script in four processes ("`mpirun -np 4 pyMPI convert.py`"). The `gridinit()` call initializes the processes grid according to the number of processes, np, determined by `mpirun`. The default option obtains a square grid or almost square, such that the number of processes in the grid is as close as possible to np. In the example of Figure 2, `mpirun` assigns $np = 4$, and the system obtains a grid configuration of 2×2 processes. However, the values of the grid configuration can be modified by the user (e.g., `gridinit(nb=2,nprow=4,npcol=1)` configures a 4×1 processes grid). Actually, some default parameters like block size in cyclic 2D distribution (nb) are fixed to an appropriated value. Optimal values for these parameters can be obtained using self-adapting software [4].

After executing `Num2PyACTS` in Figure 2, the variable `a` is a `PyACTS Array` instance in each process and its content is showed in Figure 3. Note that before the `Num2PyACTS` call only one processor (`iread==1`) has stored the array `a`. The data distribution between processors is automatically performed by that routine depending on the processes grid previously initialized and using BLACS and MPI routines in lower levels.

4. PyBLACS

PyBLACS is a Python based user friendly interface that we have developed to access the BLACS library. BLACS is a distributed communication library designed for linear algebra. The computational model consists of a one or two dimensional processes grid, where each process stores pieces of the matrices and vectors. The BLACS library includes synchronous send/receive routines to communicate a matrix or submatrix from one process to another, to broadcast submatrices to many processes, or to compute global reductions.

The names of the routines in PyBLACS are similar to the names of BLACS routines. However, in a PyBLACS routine we do not need to specify the main argument types, because PyBLACS automatically detects the correct type and calls the corresponding routine. Thus, details are hidden to

the users. In the script of Figure 4, each process sends to process [0,0] its ID; process [0,0] receives it and prints this information. This script has a similar functionality that the examples included in the standard BLACS distribution (www.netlib.org/blacs/BLACS/Examples.html). The original source Fortran code takes up around fifty lines of instructions: variable declaration and memory allocation included. On the other hand, the same functionality is coded in PyBLACS with only fourteen lines. Also, some BLACS environment parameters like `ORDER, TOP, SCOPE, ICTXT`, are hidden in the PyBLACS calls because they are not directly related to the data being processed, but more related to the computational environment and they can misguide for non advanced users.

```
from PyACTS import *
import PyACTS.PyBLACS as PyBLACS
gridinit()                              # Grid initialization
nprow,npcol,myrow,mycol=gridinfo()      # Get grid configuration
icaller = PyBLACS.pnum(myrow,mycol)     # Each process Get the own ID
if myrow==0 and mycol==0:
 for i in range[0,nprow]:
  for j in range[0,npcol]:
   if i!=0 and j!=0:
     icall=PyBLACS.gerv2d(icall,i,j)    # [0,0] receive ID from [i,j]
     print "Received ID:", icall
else:
 PyBLACS.gesd2d(icaller,0,0)            # Send ID
gridexit()
```

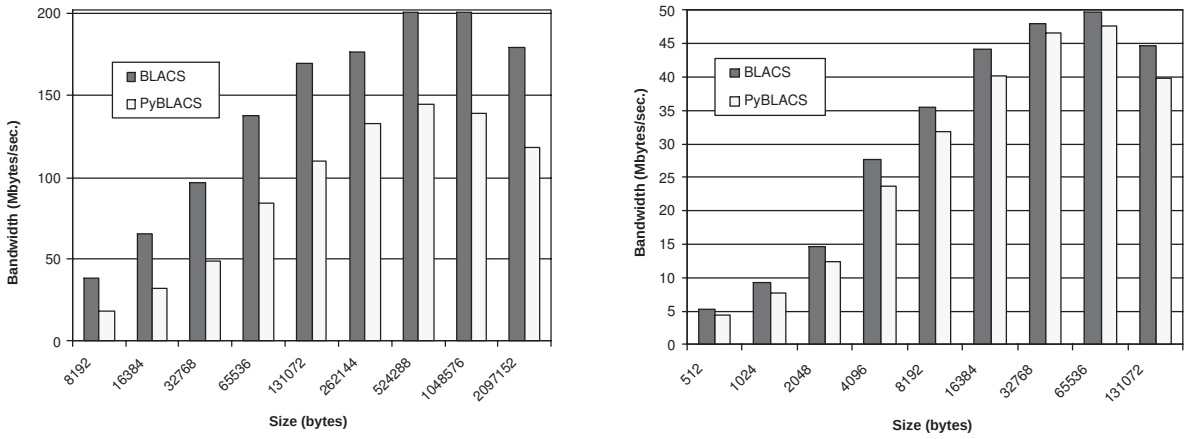
Figure 4. Example using PyBLACS.

We have implemented several tests and evaluations that compare BLACS and PyBLACS on two different computer systems. One of these is an IBM SP RS/6000, named Seaborg, located at the National Energy Research Scientific Computing Center. Seaborg is a distributed memory computer with 6080 processors. The second system that we used in our experiments is a Linux cluster (6 Intel CPU, 2 GHZ) connected with Gigabit ethernet.

One of the tests runs is shown in Figure 5. Here, we show the bandwidth obtained for different message sizes. We compare this bandwidth using the BLACS routines and the PyBLACS interfaces. In order to estimate the overhead introduced by PyBLACS, a *ping-pong* test was programmed under both Fortran and Python. Arrays of increasing size, allocated as doubles, were repeatedly sent and received. We can see that the bandwidth obtained using BLACS routines in its original form (Fortran program) is a little higher than the PyBLACS bandwidth. This is due to the fact that the Python interfaces need to check and pass arguments to the extensions (shared objects programmed in C, C++ or Fortran) [15]. However, this lost of performance is balanced with an improvement in terms of productivity, applicability and its ease of use.

5. PyPBLAS

Another high level interface we have implemented is PyPBLAS, which is a Python based interface to PBLAS [11]. PBLAS is a parallel set of BLAS routines [3]. The BLAS are high quality “building block” routines for performing basic vector and matrix operations. Level 1 BLAS do vector-vector



(a) Seaborg

(b) Linux cluster

Figure 5. BLACS and PyBLACS bandwidth comparison.

operations, Level 2 BLAS do matrix-vector operations, and Level 3 BLAS do matrix-matrix operations. BLAS and PBLAS routines are efficient, portable, and widely available. They are commonly used in the development of high quality linear algebra software.

In Figure 6, we have written a program used to test the level 3 routine (`pvgemm`). This example reads the data from the text files and store them in PyACTS Arrays. Note that this reading is done by one process (usually, [0,0] in the processes grid) and it sends the data to the rest of processes using PyBLACS to obtain a cyclic 2D distribution. After executing `Txt2PyACTS` in Figure 6, the variables `a`, `b` and `c` are PyACTS Arrays and can be used as parameters in PyACTS routines. In this example, we use the `pvgemm` routine without indicating the data types in the routine name as we must do in the PBLAS routine. This is because PBLAS routines (as well as BLACS routines) names are type-dependent, and in this case the letter '`v`' is replaced by a given data type letter (S, D, C and Z) [11]. However, as we commented in Section 4, in the Python based interfaces, it is not necessary to specify the data type because Numeric Python Arrays automatically cast them, and it is internally reused to call the corresponding routine. As we illustrate in Figure 6, we provide a set of specific tools to convert (`Scal2PyACTS`), and read data (`Txt2PyACTS`) before executing PyPBLAS routines (e.g., `pvgemm`).

For each level, we have compared the performance of both PBLAS and PyPBLAS. We have tested with different matrix and vector sizes and different number of processors and grid configuration. In Figure 7 we show the execution times for getting the solution to PDAXPY (level 1: $x + \alpha y,\ \alpha \in \mathbb{R},\ x, y \in \mathbb{R}^n$), PDGER (level 2: $\alpha x y^t + A,\ \alpha \in \mathbb{R},\ x, y \in \mathbb{R}^n,\ A \in \mathbb{R}^{n \times n}$) and PDGEMM (level 3: $\alpha AB + \beta C,\ \alpha, \beta \in \mathbb{R},\ A, B, C \in \mathbb{R}^{n \times n}$). The input data used in these operations were random vectors and matrices.

In Figures 7(a), 7(b) and 7(c), we present the execution times obtained in the cluster for the levels 1, 2 and 3, respectively. These results show that the times of the PyPBLAS routines are slightly higher than PBLAS times. Neverteless, it can be appreciated in these figures that the performance of PBLAS and PyPBLAS are similar and this shows that the overhead introduced by the Python interfaces is negligible. This fact is clearly appreciated in Figure 7(d), where we show the performance

```
from PyACTS import *
import PyACTS.PyPBLAS as PyPBLAS
ACTS_lib=1                          # ScaLAPACK ID
PyACTS.gridinit()                   # Grid initialization
alpha=Scal2PyACTS(1.2,ACTS_lib)   # Convert scalar to PyACTS scalar
beta=Scal2PyACTS(2,ACTS_lib)
a=Txt2PyACTS("data_a.txt",ACTS_lib) # Read Text file and
b=Txt2PyACTS("data_b.txt",ACTS_lib) # store in PyACTS Array
c=Txt2PyACTS("data_c.txt",ACTS_lib)
result=PyPBLAS.pvgemm(alpha,a,b,beta,c) # Call level 3 PBLAS routine
PyACTS2Text("data_result.txt",result)   # Write results to Text
PyACTS.gridexit()
```

Figure 6. Example of PyPBLAS: `pvgemm`.

obtained (in terms of MFLOPS) in the cluster using PBLAS from Fortran and from Python. As we can see, the performance is similar in both environments because intensive operations are made, in both cases, using PBLAS routines in lower levels.

References

[1] D. Ascher, P.F. Dubois, K. Hinsen, J. Hugunin, and T. Oliphant. Numerical Python. Technical Report UCRL-MA-128569, Lawrence Livermore National Laboratory, Livermore, 2001.

[2] L.S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J.W. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley. *ScaLAPACK User's Guide*. SIAM, Philadelphia, PA, 1997.

[3] L.S. Blackford, J. Demmel, J. Dongarra, I. Du, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R.C. Whaley. An updated set of Basic Linear Algebra Subroutines (BLAS). *ACM Trans. Math. Soft*, 28(2), 135–151, 2002.

[4] Z. Chen, J. Dongarra, P. Luszczek, and K. Roche. Self-adapting software for numerical linear algebra and LAPACK for clusters. *Parallel Computing*, 29(11–12):1723–1743, 2003.

[5] J. Dongarra, S. Huss-Lederman, S. Otto, M. Snir, and D. Walkel. *MPI: The complete reference*. The MIT Press, Cambridge, MA, 1998.

[6] J. Dongarra, R.C. Whaley. Basic Linear Algebra Communication Subprograms (BLACS). http://www.netlib.org/blacs

[7] L.A. Drummond, V. Hernández, O. Marques, J.E. Román, and V. Vidal. A Study of Robust Scientific Libraries for the Advancement of Sciences and Engineering. Proceedings of the 6th International Conference on High Performance Computing for Computational Science, Valencia, Spain, 2004.

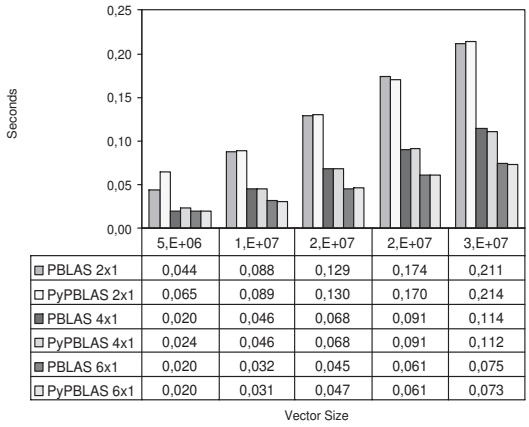
[8] L.A. Drummond and O. Marques. The ACTS Collection. Robust and high-performance tools for scientific computing: Guidelines for tool inclusion and retirement. Technical Report LBNL/PUB-3175, Computational research division, Lawrence Berkeley National Laboratory, 2002.

[9] K. Hinsen. ScientificPython User's Guide. Centre de Biophysique Moleculaire CNRS, Grenoble, France, 2002.

[10] P. Miller. PyMPI - An introduction to parallel Python using MPI. http://www.llnl.gov/computing/develop/python/pyMPI.pdf

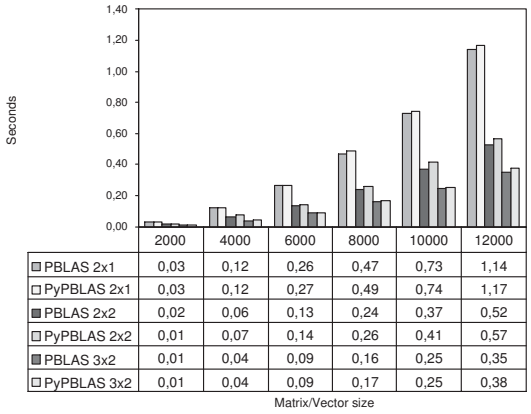
[11] NETLIB: Parallel Basic Linear Algebra Subroutines (PBLAS). http://www.netlib.org/scalapack/pblas_qref.html

[12] J. Painter and E.A. Merritt. mmLib Python toolkit for manipulating annotated structural models of



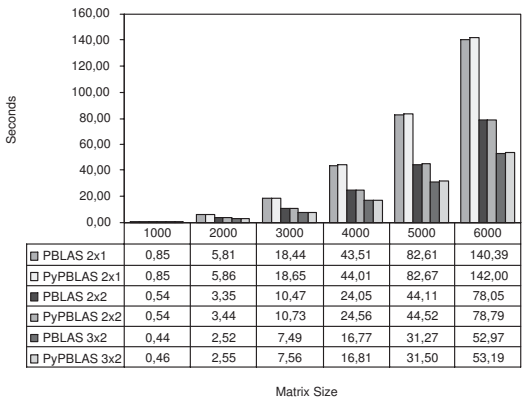
	5,E+06	1,E+07	2,E+07	2,E+07	3,E+07
PBLAS 2x1	0,044	0,088	0,129	0,174	0,211
PyPBLAS 2x1	0,065	0,089	0,130	0,170	0,214
PBLAS 4x1	0,020	0,046	0,068	0,091	0,114
PyPBLAS 4x1	0,024	0,046	0,068	0,091	0,112
PBLAS 6x1	0,020	0,032	0,045	0,061	0,075
PyPBLAS 6x1	0,020	0,031	0,047	0,061	0,073

(a) Level 1: PDAXPY. Execution Time



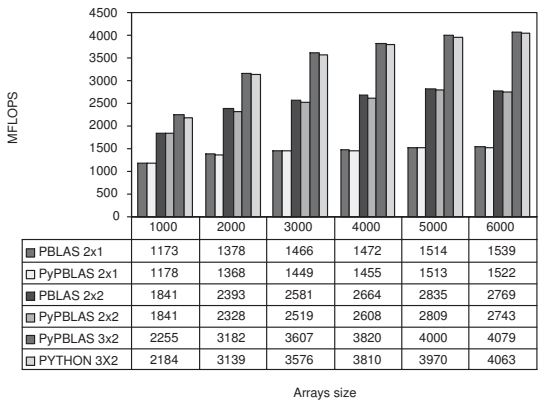
	2000	4000	6000	8000	10000	12000
PBLAS 2x1	0,03	0,12	0,26	0,47	0,73	1,14
PyPBLAS 2x1	0,03	0,12	0,27	0,49	0,74	1,17
PBLAS 2x2	0,02	0,06	0,13	0,24	0,37	0,52
PyPBLAS 2x2	0,01	0,07	0,14	0,26	0,41	0,57
PBLAS 3x2	0,01	0,04	0,09	0,16	0,25	0,35
PyPBLAS 3x2	0,01	0,04	0,09	0,17	0,25	0,38

(b) Level 2: PDGER. Execution Time



	1000	2000	3000	4000	5000	6000
PBLAS 2x1	0,85	5,81	18,44	43,51	82,61	140,39
PyPBLAS 2x1	0,85	5,86	18,65	44,01	82,67	142,00
PBLAS 2x2	0,54	3,35	10,47	24,05	44,11	78,05
PyPBLAS 2x2	0,54	3,44	10,73	24,56	44,52	78,79
PBLAS 3x2	0,44	2,52	7,49	16,77	31,27	52,97
PyPBLAS 3x2	0,46	2,55	7,56	16,81	31,50	53,19

(c) Level 3: PDGEMM. Execution Time



	1000	2000	3000	4000	5000	6000
PBLAS 2x1	1173	1378	1466	1472	1514	1539
PyPBLAS 2x1	1178	1368	1449	1455	1513	1522
PBLAS 2x2	1841	2393	2581	2664	2835	2769
PyPBLAS 2x2	1841	2328	2519	2608	2809	2743
PyPBLAS 3x2	2255	3182	3607	3820	4000	4079
PYTHON 3X2	2184	3139	3576	3810	3970	4063

(d) Level 3: PDGEMM. MFLOPS.

Figure 7. PBLAS versus PyPBLAS.

biological macromolecules. *Journal of Applied Crystallography*, 37:174–178, Part 1, 2004.

[13] J. Sáenz, J. Zubillaga, and J. Fernández. Geophysical data analysis using Python. *Computers & Geosciences*, 24(4):457–465, 2002.

[14] G. van Rossum and F.L. Drake Jr. *An Introduction to Python*. Network Theory Ltd, 2003.

[15] G. van Rossum and F.L. Drake Jr. *Extending and Embedding the Python Interpreter*. Network Theory Ltd, 2003.

Parallelization of GSL on Clusters of Symmetric Multiprocessors*

J. Aliaga[a], F. Almeida[b], J.M. Badía[a] , S. Barrachina[a], V. Blanco[b], M. Castillo[a], R. Mayo[a], E.S. Quintana[a], G. Quintana[a], C. Rodríguez[b], F. de Sande[b], A. Santos[b]

[a]Depto. de Ingeniería y Ciencia de Computadores, Univ. Jaume I, 12.071–Castellón, Spain; {aliaga,badia,barrachi,castillo,mayo,quintana,gquintan}@icc.uji.es.

[b]Depto. de Estadística, Investigación Operativa y Computación, Univ. de La Laguna, 38.271–La Laguna, Spain; {falmeida,vblanco,casiano,fsande}@ull.es.

In this paper we discuss the application of an hybrid programming paradigm that combines message-passing (MPI) with shared memory programming (OpenMP). We apply this model to the parallel solution of two basic problems: the sparse matrix-vector product and the dynamic programming problem. We compare the results of the hybrid model with the application of a pure MPI model on a cluster of dual Intel Xeon processors. The experimental results show that the behavior of both models depend, among other factors, on the application and on the size of the problems. While with the dynamic programming problem we obtain very good speedups, in the case of the matrix-vector product the algorithms do not take very good profit of the dual processors.

1. Introduction

The GNU Scientific Library (GSL) [6] is a collection of hundreds of routines for numerical scientific computations written in ANSI C, which includes codes for complex arithmetic, matrices and vectors, linear algebra, integration, statistics, and optimization, among others. The reason GSL was never parallelized seems to be the lack of a globally accepted standard for developing parallel applications. We believe that with the introduction of OpenMP and MPI the situation has changed substantially. OpenMP [8] has become a standard *de facto* for exploiting parallelism using *threads*, while MPI [7] is nowadays accepted as the standard interface for developing parallel applications following the message-passing programming model.

On the other hand, the application of parallelism has increased in recent years partly due to the arising of the clusters of personal computers. This kind of parallel architecture can be easily upgraded and offers a very good performance/price relation. Clusters are naturally programmed using the message-passing model. However, most recent clusters are composed of nodes that are small shared-memory multiprocessors containing from 2 to 8 standard personal computers. A trend of these hybrid parallel computers could be the inclusion of multi-core processors, while SMPs are being clustered to increase the number of CPUs of the architecture.

It seems that a natural, efficient and portable way to take profit of hybrid architectures is to combine the MPI library to communicate the different nodes, with OpenMP to run several threads on the processors included in each node of the cluster [11], [3], [1].

This hybrid paradigm allows us to exploit efficiently the shared memory without having to add MPI communications within each node. Besides, this paradigm could combine coarse grain parallelism among the nodes with fine grain parallelism within them, exploiting besides the dynamic load balancing possibilities of the OpenMP model. However, there is not guarantee that the hy-

*This work has been supported by the EC (FEDER) and the Spanish MCyT (Plan Nacional de I+D+I, TIC2002-04400-C03-03 and TIN2005-09037-C02).

brid model obtains better performance that a pure MPI implementation, [2]. Besides, other parallel programming models can also be used on clusters of SMPs [9].

In this paper we investigate the parallelization of a specific part of GSL using the hybrid parallel model, and we compare it with the use of a pure message-passing model based on the MPI library. In Section 2 we elaborate our first case study, describing a parallelization of the sparse matrix-vector product addressed to SMPs, and the experimental results. In Section 3 we perform the same kind of description and analysis in the case of the dynamic programming problem. Finally, concluding remarks follow in Section 4.

2. Case study I: sparse matrix-vector product

Sparse matrices arise in a vast amount of areas, some as different as structural analysis, pattern matching, control of processes, tomography, or chemistry applications, to name a few. Surprisingly enough, GSL does not include routines for sparse linear algebra computations. This can only be explained by the painful lack of standards in this area: Only very recently the BLAS Technical Forum came with a standard [5] for the interface design of the *Basic Linear Algebra Subprograms* (BLAS) for unstructured sparse matrices.

The implementation, parallelization, and performance of sparse computations strongly depend on the storage scheme employed for the sparse matrix which, in many cases, is dictated by the application the data arises in.

Two of the most widely-used storage schemes for sparse matrices are the coordinate and the Harwell-Boeing (or compressed sparse array) formats [4]. As there seems to be no definitive advantage of any of the above-mentioned schemes, in our codes we employ a variant of the rowwise coordinate format.

Our approach to deal with parallel platforms consists in dividing the matrix into p blocks of rows with, approximately, the same size. Each process operates then with the elements of its corresponding block of rows.

The following (simplified) data structure is used to manage distributed sparse matrices on each processor:

```
1: typedef struct {
2:   size_t local_size1;   // local row size
3:   size_t local_size2;   // Global column size
4:   size_t local_nz;      // Global non-zeros
5:   int **rowptr; // pointers to rows
6:   void *local_data;
7: } internal_dmdd_sparse_matrix;
```

In this structure, `rowptr` stores pointers to the init of each row in the vector `local_data`. This vector stores the matrix elements in the processor by rows. Each element is stored as three adjacent values: `(row_index, column_index, element_value)`. We have chosen this storage scheme instead of the three vectors of the classical coordinate format in order to improve the locality in the access to the elements of the matrix.

2.1. Parallelizing the Sparse Matrix-Vector Product

We describe next the parallel implementation of the sparse matrix-vector

$$y \leftarrow y + \alpha \cdot A \cdot x,$$

where a (dense) vector y, of length m, is updated with the product of an $m \times n$ sparse matrix A times a (dense) vector x, with n elements, scaled by a value α. Notice that this is by far the most common operation arising in sparse linear algebra [10], as it preserves sparsity of A while allowing to employ codes that exploit the zeroes in the matrix to reduce the computational cost of the operation. For simplicity, we ignore hereafter the more general case, where A can be replaced in (2.1) by its transpose or its conjugate transpose.

First we will describe the parallelization of the product using a message-passing model, and then we will describe how we can modify it so that each process executes several threads on a shared-memory environment.

The (sparse) matrix-vector product is usually implemented as a sequence of *saxpy* operations or dot products, with one of them being preferred over the other depending on the target architecture and, in sparse algebra, the specifics of the data storage.

In our approach, taking into account the rowwise storage of the data, we decided to implement the parallel sparse matrix-vector product as a sequence of dot products. In order to describe the code we assume that the vector x involved in the product is initially replicated by all processes. We also consider a block partitioning of vector $y = (y_0, y_1, \ldots, y_{p-1})$, with approximately an equal number of elements per block, and a partition of the sparse matrix A by blocks of roughly m/p rows as $A = \left(A_0^T, A_1^T, \ldots, A_{p-1}^T\right)^T$.

The parallelism of the rowwise version of the product arises from the fact that every dot product among a row of the distributed matrix A and the replied vector x can be performed fully in parallel to obtain different elements of the solution vector y. Therefore, the code executed on each process in the pure message-passing version of the product is the following:

```
1:  for (i = 0; i < local_size1; i++) {
2:    pos = rowptr[i];  // first element in row i
3:    k = row_index(pos);
4:    temp = 0.0;
5:    for (j =  0; j < nz_in_row(i); j++) {
6:      temp += value_in(pos) * x[k];
7:      inc_coordinate(pos);
8:    }
9:    y[k] += alpha * temp;
10:  }
```

In the previous code, `row_index(pos)` is a macro that obtains the row index of the matrix element from the position `pos` in `local_data`. Another C macro, `nz_in_row(i)`, returns the number of non-zero elements in the `i-th` row. Finally, `inc_coordinate(pos)` shifts the position `pos` to the next element of the matrix.

Once each process has computed a block of the solution vector y, a collective communication (of type `MPI_Allgather`) is required to replicate the results. This operation is (almost) perfectly balanced as all processes have a close number of elements of y.

Let us now describe the hybrid version of the algorithm. Suppose that each MPI process is executed on a node that is a shared memory multiprocessor. An easy and portable way to take profit of this kind of architecture is to use OpenMP in order to distribute the computations of each process among the different processors of each node. Exploiting the same idea about the parallelism of the matrix-vector product than before, we can compute in parallel the iterations of the outer loop of the previous algorithm, corresponding to independent dot products. This idea can be implemented by adding the following directive:

```
#pragma omp parallel for private (pos, k, temp, j)
```

just before the outer loop. A certain amount of dot products are thus computed by each thread and the threads only need to be synchronized once, on termination of the outer loop. If the non-zero elements of the matrix are not evenly distributed among the different rows, a dynamic scheduling of the threads provides a better load balancing. This could be one of the main advantages of using OpenMP threads on each node instead of MPI processes, where load balancing should be implemented by the programmer by means of an appropriate data and task distribution.

It is worth noticing that the global communication among the MPI processes is performed once finished the OpenMP parallel region. The different threads have partially contributed to the computation of the solution vector corresponding to the each MPI process. Then, the master thread on each node can access that information and perform the global communication. Since the communication is out of the OpenMP parallel region, theoretically a non-thread safe version of MPI could be used.

2.2. Experimental analysis

Our experimental platform is a dual-SMP cluster of 34 nodes. Each node consists of two Intel Xeon Processors at 2.4 Ghz., 1GB of RAM and 512 KB of L2 cache. The nodes are connected through a Myrinet network with a bandwidth of 2Gb/s. Each node runs a Linux kernel version 2.4.24bi and we used Intel icc compiler version 7.1 with the option -O3. The algorithms have been implemented with the MPI v.1.2.5 compiled with the `-ch_gm` option in order to exploit the GM communication library of the Myrinet network.

We have performed experiments with three versions of the parallel matrix-vector product:

- An *uniprocessor* version in which we run the pure MPI algorithm using only one of the processors of each node.
- A *pure MPI* version in which we have run the same pure MPI algorithm spawning two MPI processes on each node. In order to exploit the shared memory for the intra-node communications we used the option `--gm-numa-shmem` of the `mpirun.ch_gm` command.
- A *hybrid MPI+OpenMP* version in which we run a MPI process per node and this process spawns two OpenMP threads to perform the computation.

We have tested the performance of the three versions with different problem sizes (sparsity and matrix size) and using different number of processors. The left-hand side of Fig. 1 shows the speed-ups of the parallel algorithms with respect to the sequential algorithm. The figure shows the results for matrices of two sizes: 30.000 (30k) and 40.000 (40k), both cases with an sparsity of 2%.

The speed-ups are far from the optimal values. This is mainly due to the effect of the communication cost in an application with a computation cost depending on a small number of non-zero elements. On the other hand, as we use random generated matrices, the number of non-zero elements on each processor is balanced, and so the number of flops. However the distribution of the elements in the columns is quite different on the different processors, producing a different access pattern to the memory and unbalancing the total computational cost.

The experiments show also that the speed-up of the hybrid version overcomes the speed-up of the uniprocessor version when we take profit of the dual nodes of the machine. However the pure MPI version of this application in our experimental environment is only better than the uniprocessor version for some problem sizes, for example, matrices with 40k rows. We can also see that, as we increase the size of the matrix the speed-up of the pure MPI version approaches the speed-up of

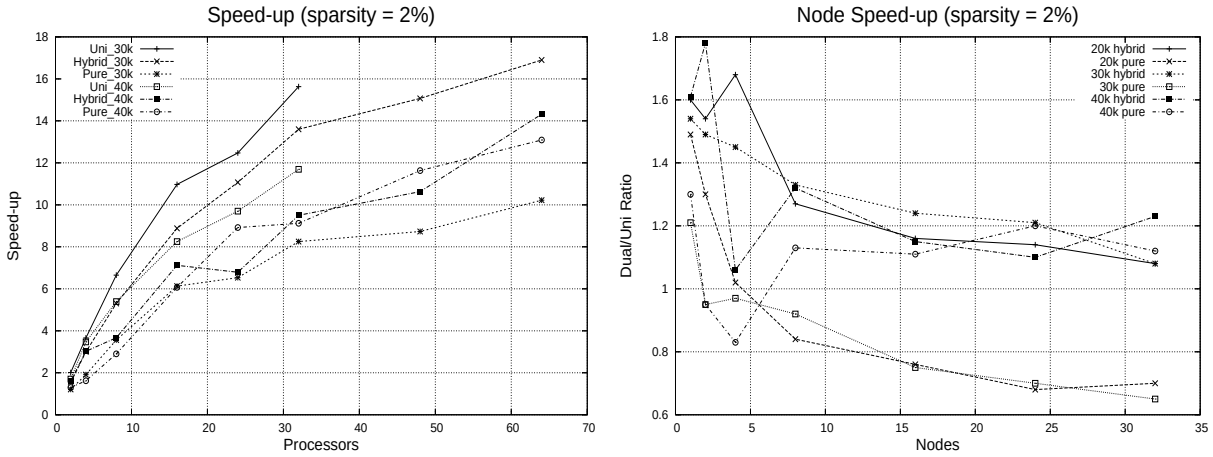


Figure 1. Performance of the parallel versions of the sparse matrix-vector product.

the hybrid version, and in some cases even surpasses it. In order to analyze the use of the hybrid architecture we plot in the right-hand side of Fig. 1 the *Node speed-up* [2]. Given a fixed number of nodes, this parameter is the ratio between the time spent using the two processors on each node and using only one processor per node. The figure shows that the results obtained are always suboptimal (< 2). In some cases, the pure MPI versions offers node speed-ups less than 1, which means that it would have been better to run only one process per node than to run two processes trying to use both processors.

We could expect that, for a fixed number of processors, the uniprocessor version offers better results than the pure and hybrid versions, because in the last two cases the two processes or threads running on each node have to share its resources (memory, network, etc.). However, the results shown in Fig. 1 are worse than expected even having into account the previous kind of collision. The detailed justification of the experimental results requires a deeper analysis of the influence of several factors, including the communication and computation costs, the use and access to the memory, etc. We intend to develop this analysis of the application and the experimental environment in a near future.

3. Case study II: dynamic programming

Dynamic programming (DP) is an important problem-solving technique that has been widely used in various fields such as control theory, operations research, biology and computer science. Dynamic programming is a useful method when the solution of a problem can be expressed as the result of a sequence of decisions. The method enumerates all decision sequences and selects the best among them. Dynamic Programming often reduces the amount of enumeration drastically by avoiding the consideration of decision sequences that will not deliver optimal solutions. An optimal sequence of decisions is obtained by making explicit use of the principle of optimality. This approach derivates into a general recurrence formula where, on a multistage-problem, the optimal values for subproblems involving i decisions are computed in terms of subproblems involving $i - 1$ decisions.

For instance, in the Single Resource Allocation Problem, we need to allocate M units of an indivisible resource to N tasks so that the sum of the effectiveness is maximized. The problem can be

formally stated as:

$$\max\ z\ = \sum_{j=1}^{N} f_j(x_j) \quad \text{subject to } \sum_{j=1}^{N} x_j = M,$$

where $f_j(x_j)$ gives the benefit of allocating x_j units of resource to task j.

Now denote by $G[i][x]$ the optimal benefit of the subproblem and consider the first i tasks and x units of resource. The dynamic programming recurrence equations are then formulated as follows:

$$\begin{array}{rcl} G[i][x] & = & \max\{G[i-1][x-j] + f_i(j) : 0 < j \leq x\}, \quad i = 2, ..., N, \\ G[1][x] & = & f_1(x), \quad 0 < x \leq M, \quad \text{and} \\ G[i][x] & = & 0, \quad i = 1, ..., N; \quad x = 0. \end{array}$$

The value $G[N][M]$ gives the total income for the optimization problem and, in order to be computed, the dynamic programming table G is needed in advance.

3.1. Parallelizing the Dynamic Programming Problem

The Dynamic Programming table is necessary to obtain the optimal solution, and the values of the table must be computed following the order imposed by the dependences of the recurrence equations. A simple parallelization of this operation on a distributed-memory platform using MPI replicates the table on all nodes. With G distributed by columns, the processes compute in parallel the entries belonging to the same row of the table, with the rows being computed sequentially from top to bottom. We next show the code executed on the $myid$ MPI process:

```
1:   for (i = 0; i <= N_TASKS; i++) {
2:      for (x = displs[myid]; x < displs[myid + 1]; x++) {
3:         G[i][x] = (*f)(i, 0);
4:         for (j = 0; j <= x; j++) {
5:            fix = G[i - 1][x - j] + (*f)(i, j);
6:            if (G[i][x] > fix)
7:               G[i][x] = fix;
8:         }
9:      }
10:     MPI_Allgatherv(&G[i][displs[myid]], ... );
11:  }
```

An *all-to-all* communication operation is performed to replicate every row. Since the inner loop ($0 < j \leq x$) is not constant, we follow a block distribution with variable block sizes. The sizes of the blocks are computed so that they minimize the load imbalance. The number of columns assigned to each processor are computed as an arithmetic progression, and the starting indexes are stored in vector `displs`. This evaluation also forces the use of different parallel regions in the OpenMP version.

The following code summarizes the hybrid version of the algorithm.

```
1:   #pragma omp parallel private(...)
2:   for (i = 0; i <= N_TASKS; i++)
3:      for (x = th_displs[th_id]; x < th_displs[th_id + 1]; x++)
4:           // idem than in the pure MPI version
5:      #pragma omp barrier
6:      #pragma omp master
7:        MPI_Allgatherv(&G[i][th_displs[th_id]], ... );
```

The hybrid algorithm exploits the same kind of parallelism than the pure MPI version on each node. The columns assigned to each MPI process, and so the iterations of the loop on line 3, are distributed among several OpenMP threads. In order to balance the load we can distribute a different number of columns to each thread following the same kind of distribution than in the pure MPI version. The starting column assigned to each thread is stored in vector `th_displs`.

On the other hand, each time the algorithm finishes the computations corresponding to one of the rows of the matrix, the MPI processes have to perform an *all-to-all* communication operation. This communication can only be performed when all the threads have finished their iterations of the loop on line 3, and so we added an OpenMP barrier on line 5. Besides, we used a `omp master` directive to guarantee that only the master thread participates on the MPI communication.

3.2. Experimental analysis

In order to perform the experimental analysis of this second case of study we used the same experimental environment that with the sparse matrix-vector product. We tested the same three versions of the algorithm (uniprocessor, pure MPI and hybrid MPI + OpenMP) with different problem sizes: 1000 (1k) and 3000 (3k).

The left-hand side of Fig. 2 reports the speed-ups of the three versions of the algorithm. The speed-ups are very good for all the versions of the algorithm and they increase with the size of the problem. We can even observe super-linear speed-ups with the largest problem, that are probably a consequence of a better use of the cache memory in the parallel algorithms.

Regarding the relative behavior of the three versions of the algorithm, we can see that both, the pure MPI and the hybrid version, continue the almost linear increasing of the speed-ups of the uniprocessor version when we exploit both processors of the nodes. The right-hand side of the figure 2 shows that the node speed-ups are always larger than 1 and in some cases even larger than 2. Moreover, the node speed-ups increase with the problem size, and they are almost independent of the number of nodes in the case of the largest problem size.

Besides, for the largest problem size, the hybrid parallelization offers better speed-ups than the pure MPI one. The performance behavior, is reversed for the smallest problem size where speed-ups of the pure MPI code surpass the speed-ups of the hybrid version.

Although even better performances could be achieved using tuned tiled block-cyclic parallelizations, this requires information on the optimal tile sizes a priori. Since these sizes are architectural and problem dependent, the approach seems to be hardly suitable for a general purpose library.

4. Conclusions

In this paper we study different models to parallelize algorithms on clusters of SMPs. Specifically we study two basic models: a pure MPI model and a hybrid MPI + OpenMP model. In the first case a MPI process is executed on each processor. In this second model an MPI process is executed on each node and OpenMP threads cooperate to perform the task assigned to each process.

In order to compare both models we used two applications: the sparse matrix-vector product and the dynamic programming problem. Both applications use different schemes of parallelization. In the matrix-vector product the algorithm starts by computing the product without communications, and then an all-to-all communication is performed to gather the result in all the processes. The dynamic programming problem is solved by means of succession of steps. Each one starts with a computation phase and finishes with a communication phase that synchronizes all the processes.

The experimental results on a cluster of dual processors show that the performance mainly depends on the application and on the problem size. While in the case of the dynamic programming problem

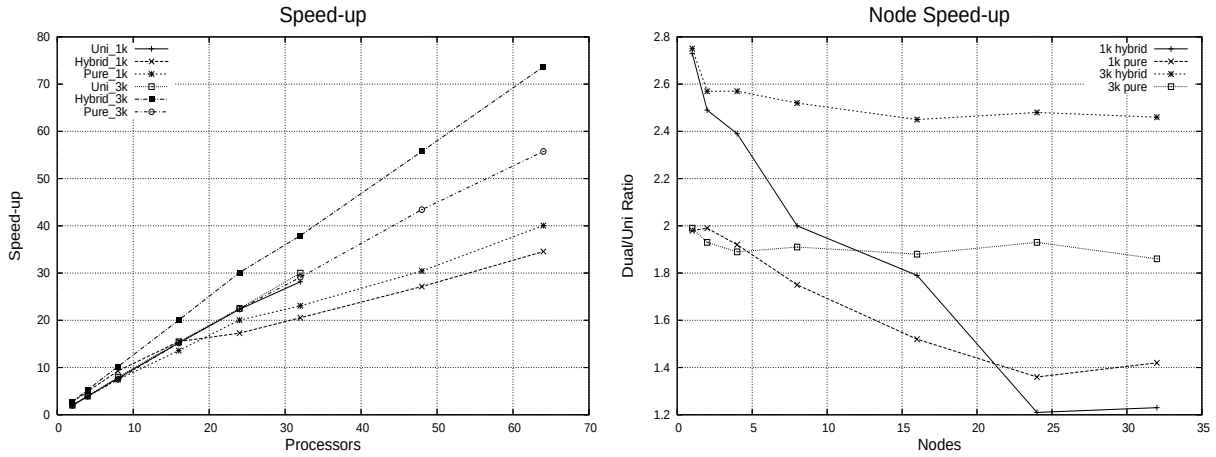


Figure 2. Performance of the parallel versions of the dynamic programming problem.

we obtain very good speed-ups, the results are not so good in the case of the matrix-vector product. In the first case both parallel models take profit of the dual node clearly increasing the speed-ups of the algorithm executed on one processor of each node. However, in the case of the matrix-vector product the results using both processors per node are better than the results using one processor per node only in some cases. Finally, the programming model that obtains the best results depend in both applications on the size of the problem.

References

[1] Franck Cappello and Daniel Etiemble. MPI versus MPI+OpenMP on IBM SP for the NAS Benchmarks. In *Supercomputing'00*, 2000.

[2] Franck Cappello, Olivier Richard, and Daniel Etiemble. Investigating the performance of two programming models for clusters of SMP PCs. In *6th IEEE Symp. on High-Performance Computer Architecture*, pages 349–359, 2000.

[3] N. Drosinos and N. Koziris. Performance comparison of pure MPI vs hybrid MPI-OpenMP parallelization models on SMP clusters. In *18th Int. Parallel & Distributed Symposium*, pages 15 (CD–ROM), 2004.

[4] I.S. Duff, A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices*. Clarendon Press, Oxford, UK, 1986.

[5] I.S. Duff, M.A. Heroux, and R. Pozo. An overview of the sparse basic linear algebra subprograms. *ACM Trans. Math. Software*, 28(2):239–267, 2002.

[6] M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, M. Booth, and F. Rossi. *GNU scientific library reference manual*, July 2002. Ed. 1.2, for GSL Version 1.2.

[7] MPI Forum web page. Available at *http://www.mpi-forum.org*.

[8] OpenMP web page. Available at *http://www.openmp.org*.

[9] R. Rabenseifner. Hybrid parallel programming: Performance problems and chances. In *45th CUG Conference*, pages 12–16, 2003.

[10] Y. Saad. Iterative methods for sparse linear systems. 2nd edition with corrections; available at *http://www-users-cs.umn.edu/˜saad/books.html*, January 2000.

[11] Lorna Smith and Mark Bull. Development of mixed mode MPI / OpenMP applications. *Scientific Programming*, 9(2-3/2001):83–98, 2001. Presented at Workshop on OpenMP Applications and Tools (WOMPAT 2000), San Diego, Calif., July 6-7, 2000.

Simulations

Parallel Program Complex for High Reynolds Unsteady Flow Simulation

Boris N. Chetverushkin[a], Eugene V. Shilnikov[a]

[a]Institute for Mathematical Modeling Russian Academy of Sciences, 125047 Moscow, Russia

1. Introduction

Essentially unsteady and turbulent regimes of viscous gas flows have received increasing attention from researches, motivated in part from the importance of unsteadiness in industrial problems arising in turbomachinery and aeronautics. First of all this is connected with the possible destructive influence of the acoustic pressure oscillations occurring in the gas flow upon mechanical properties of the construction elements, especially in the resonant case. Under certain conditions such flows may be characterized by quasi-regular self-induced pressure oscillations. Their frequency, amplitude and harmonic properties depend upon the body geometry and external flow conditions. Such unsteady flow phenomena which occur frequently behind relatively slender, bluff structures are of great practical interest. In the case of symmetric geometry at relatively small Reynolds numbers numerical simulation based on 2D unsteady Navier - Stokes equations are quite successful. At high Reynolds numbers, which are more relevant in practice, three-dimensional stochastic turbulent fluctuations are superimposed on the quasi-periodic 2D unsteady motion. So numerical simulation must be three-dimensional even for simple flow geometry. The numerical simulation of a detailed structure of unsteady viscous compressible 3D flows with high Reynolds numbers is very expensive. It is possible only by use of high performance parallel computer systems. This demands the development of the specialized parallel software. The development of such software is one of the goals of our activity. This software must have a good scalability (with respect as to the number of processors as to the problem size), portability and robustness.

This paper concerns with the future development of a parallel MPI-based program package for 3D viscous gas flow simulation presented in [1]. One of the restrictions of this package was its orientation on using only structured non-uniform meshes. This restriction may be rather taxing when high accuracy resolution of the solution particularities in small regions is needed. Such situations frequently occur during numerical modeling a large amount of modern problems of mathematical physics. These peculiarities may be the result of as physical processes as problem geometry (for example, detonation or combustion processes, solitons motion, body shape singularity etc.). Using of regular non-uniform meshes in such situation leads to inevitable spreading of fine mesh into regions where it is not necessary. Fig. 1 demonstrates an example of such mesh for simulation of viscous gas flow around a solid body. We do not need the fine grid along lines BE and BD. So we will have a waste of computational time, especially in 3D case. This problem may be avoided by use of unstructured meshes, but the convenience and simplicity of difference schemes on regular grids enforced us to search a way of solving a problem in the frames of structured meshes. It is naturally in this situation to use multiblock grids where different subregions have their own grids. A variant of such approach is the use of nested (or locally refined) grids (see Fig. 2).

Parallel realization of explicit finite difference schemes for the numerical solution of gas dynamic problems on nested grids were presented in [2]. The parallel program complex [1] was supplemented with the possibility of rectangular locally refined grids usage. Special utility for such grid partitioning was created based on the multistep algorithm described in [2]. The modified program complex presented here was tested on the problem of vortex shedding past a square cylinder. Results

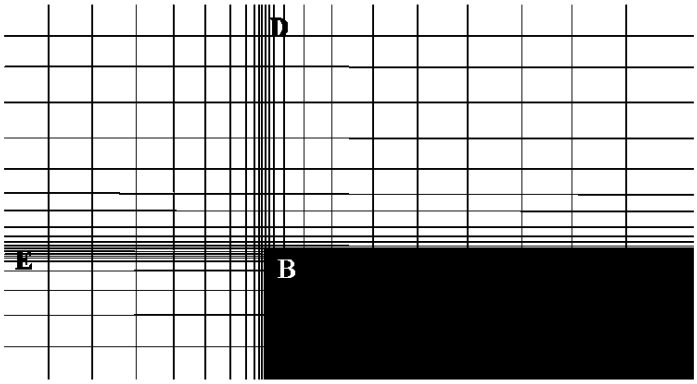


Figure 1. Sample of rectangular grid

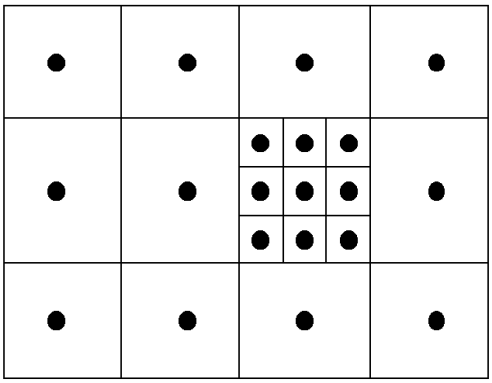


Figure 2. Locally refined grid

of simulation were compared with the experimental data [3], [4] and numerical results [5].

2. Numerical Method

Use of parallel computer systems with distributed memory architecture determines the choice of numerical method. The opinion is widely spread that we have to use only implicit schemes for viscous gas flow simulation because of their good stability properties. In the case of essentially unsteady flow we have to receive detailed information about high frequency oscillations of gas dynamic parameters. This fact limits the time step acceptable for the difference scheme by the accuracy requirements. For many interesting problems these limitations neutralize the advantages of implicit schemes. So for such problems the explicit difference schemes seem to be preferable because of their simplicity for program realization, especially for parallel implementation. Our program complex is based on explicit finite difference schemes, which are constructed as an approximation of conservation laws (control volume method). The explicit kinetically consistent finite difference (KCFD) schemes [6] were selected for realization. They have soft stability condition ($\tau \sim h$) giving the opportunity to use very fine meshes to study the fine flow structure.

KCFD schemes belong to the class of kinetic schemes. Nowadays the kinetic or Boltzmann schemes are very popular in computational fluid dynamics [7] – [9]. The kinetic schemes differ from the other algorithms primarily in that the basis for their foundation is some discrete model for one-particle distribution function. The averaging of this discrete model with the components of the collision vector leads to discrete equations for gas dynamic parameters. The successful experience in solving various gas dynamic problems by means of KCFD schemes showed that they describe viscous heat conducting flows as good as schemes for Navier-Stokes equations, where the latter are applicable. In addition to this KCFD schemes permit to calculate oscillating regimes in super- and transonic gas flows, which are very difficult for modeling by means of other algorithms. Additional dissipative terms in KCFD schemes are small in comparison with the terms of natural viscosity and conductivity and can be interpreted as efficient numerical stabilizers, which provide the solution smoothing on the distances of the order of free path length.

3. Parallel Implementation

The basic idea for the program complex constructing was to simplify ultimately the parallel program providing its lucidity. It follows thence that all complicated logical communications and other

sophisticated but not time-consuming operations should be addressed to sequential preprocessing utilities as far as possible. The program complex structure is defined by four relatively independent tasks:

- problem/model formulation including description of computation field geometry, initial and boundary conditions specification and grid generation;
- grid partitioning and data preprocessing;
- main computational block execution;
- data postprocessing.

As a rule only third of these tasks must be parallel. If the problem size is very large the data postprocessing and results visualization may be parallel too. But these problems are beyond this paper. Separate sequential programs may execute all other jobs.

Parallel realization is based on geometry parallelism principle. This means that each processor provides calculation in its own subdomain. In the case when the difference scheme is written in the form of conservation laws, the approximation of gas dynamic equations comes to approximation of conservative variables (density ρ, momentum $\rho\mathbf{U}$ and total energy E) fluxes through cell faces. In order to reach the algorithm homogeneity the boundary conditions of different types (no slip, symmetry, impermeability, inlet, outlet conditions etc.) are also written as fluxes of conservative variables through region bound. Each cell face is supplied by an attribute indicating its type: inner face, various boundary faces, face between cells of different size (result of local mesh refining), ghost face i.e. face between ghost cells. This attribute determines which subroutine must calculate fluxes through the face. So the face with its attribute becomes the basic geometrical object. The face attributes as well as description of problem geometry and grid information are contained in a special text file, which is prepared by sequential preprocessing utility. Another utility divides 3D computational region with rectangular bounds (in i-j-k space) into required number of rectangular subdomains according to multistep algorithm described in [2]. As a result this utility creates a text file describing 3D subregions in terms of grid node numbers, list of neighbors for each subregion and information needed for organizing of inter processor communications. User can edit these configuration files manually if needed.

According to this the main computational module of parallel program calculates fluxes through all faces in a subregion addressed to each processor. Having equal number of nodes in each subdomain the homogeneity of algorithm automatically provides load balancing of processors. The explicit form of schemes allows to minimize the information exchange. Note, that such program construction permits to introduce new types of boundary conditions and change as finite difference scheme as governing equation system (changing the coordinate system for example). Introducing new face types and writing new subroutines for flux calculations may reach it. This opens wide perspectives for the further development of program complex presented.

C and Fortran languages were used to develop program package. MPI libraries were taken for realization of data exchange among processors.

4. Test Problem Simulation

A program complex was tested on a problem of subsonic ($M_\infty = 0.135$) viscous compressible gas flow around square cylinder at $Re = 20000$. The inlet of the computational region was located at

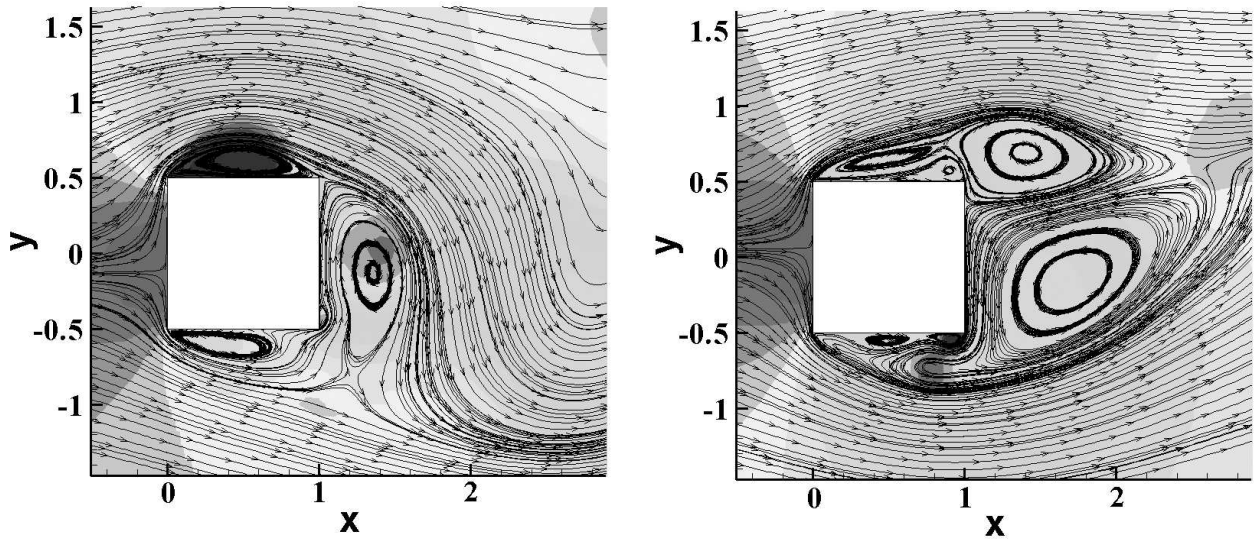


Figure 3. Stream traces in $z = 0$ section

$x/D = -5$ and outlet at $x/D = 15$, where $D = 1$ is cylinder diameter, the origin of the coordinate system was in the center of the cylinder front surface. The hight and width of the computational region were equal to $20D$. Locally refining computational grid was used. The original coarse grid with maximum cell size equal to 0.2 was refined near cylinder walls so that the minimum cell size was 0.001. Total amount of grid cells was about 8000000. The no slip conditions were posed on cylinder walls. The far field boundary conditions based on splitting of the convective fluxes by Steger – Warming [10] were used. At the beginning of calculations the symmetric flow is formed, but after a while this symmetry is broken due to vortex shedding and the flow becomes quasi-periodic. The stream traces near the cylinder for different flow phases are shown in Fig. 3. The background there is the pressure distribution. Pictures on this Figure are quite nutural for the flow type under consideration and are similar to ones, presented in [3], [4], where detailed experimental data for such flow are described.

In [5] the results of numerical simulation of the case studied in [4] are presented. The comparison of calculation results obtained with different turbulence models in 2D problem formulation was held. Our results are in a good agreement with these results. The comparison of calculated and experimental integral characteristics such as mean value of drag coefficient $\overline{c_D}$, amplitude of middle-range oscillations of lift coefficient c'_L and Strouhal number Sh corresponding to these oscillations are summarized in Table 1.

Table 1
Force coefficients and Strouhal numbers

	$\overline{c_D}$	c'_L	Sh
Experimental results [4]	2.05 – 2.19	—	0.135 –0.139
Calculated results [5]	1.56 – 2.11	0.30 – 1.18	0.129 – 0.146
Our results	2.09	0.37	0.137

It is necessary to note, that our results were obtained without introducing any turbulence model. This fact gives us the hope to use the program complex presented for direct turbulence simulation on fine space grids.

The amplitude spectrum of lift coefficient oscillations is presented in Fig. 4. The largest peak on the plot corresponds to the periodic component of the flow.

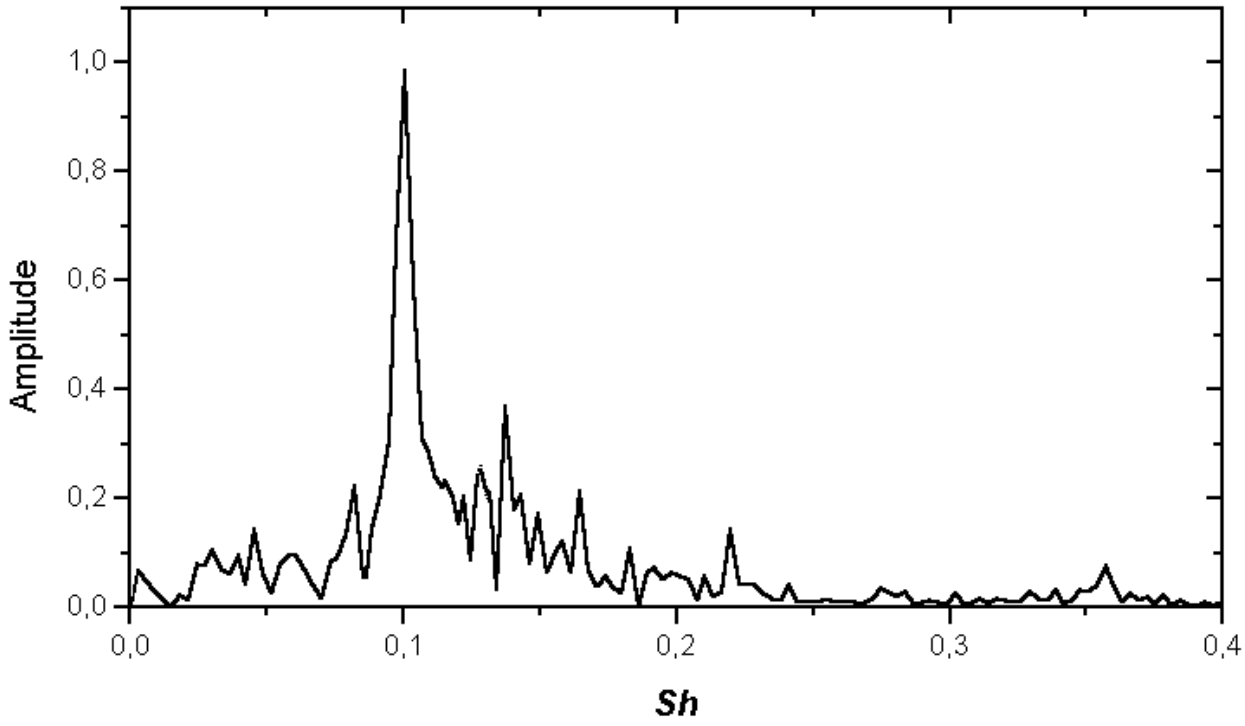


Figure 4. Spectrum of lift coefficient

5. Scalability Investigation

The scalability of program complex was investigated on 3D subsonic and supersonic problems. The calculations were held on different cluster type multiprocessor computer systems (768-processor MCS-1000M MPP computer system equipped with 667MHz 64-bit 21164 EV67 Alpha processors and 906-processor MCS-15000 MPP computer system equipped with 2.2GHz PPC970FX processors). The parallelization efficiency was measured for different number of processors (up to 600). It is more than 90% for 40 processors and more than 60% for 600 processors. The results can be found in Table 2.

Because of lack of memory such a large problem (8×10^6 cells) can't be solved on one processor of MCS-1000M system. That is why the efficiency for this system was computed with respect to the calculations held on 2 processors.

The scaling with respect to the problem size was also inspected. Our grid was doubled in each direction, so total number of cells achieved $\sim 6 \times 10^7$. Such problem is too large even for one processor of MCS-15000 system. That's why instead of efficiency calculation the times were compared

Table 2
Efficiency (%) dependence on the number of processors

N	1	2	10	40	160	320	600
MCS-1000M	—	100	97	92.8	79.9	70.1	61.4
MCS-15000	100	98.5	96.3	92	80.1	72.1	62.7

needed for 2000 time steps of our program on these two grids. The efficiencies for small and large grids are

$$\eta^{(s)} = \frac{t_1^{(s)}}{nt_n^{(s)}} \qquad \text{and} \qquad \eta^{(l)} = \frac{t_1^{(l)}}{nt_n^{(l)}}.$$

We don't know the times $t_1^{(s)}$ and $t_1^{(l)}$, but for homogeneous algorithm these times must differ by the factor of 8 from each other, i.e. $t_1^{(l)}/t_1^{(s)} = 8$. From these equalities the following formula can be found

$$\frac{\eta^{(l)}}{\eta^{(s)}} = \frac{8}{f}, \qquad \text{where} \qquad f = \frac{t_n^{(l)}}{t_n^{(s)}}$$

If the efficiency doesn't depend on the problem size the factor f must be approximately 8. The diminishing of this factor corresponds to the efficiency increase for large problem. The values of this factor for some numbers (N) of processors are presented in Table 3.

Table 3
Computational time increase for enlarged grid

N	10	40	100	200	400	600
f	7.96	7.83	7.71	7.48	7.13	6.69

These results show that for $N < 100$ our factor is really close to 8, but for greater N it diminishes. This effect is connected with the increase of computational time with respect to exchange time for each processor. So, increasing of the total grid nodes number leads to the efficiency growth. For example factor 6.69 for $N = 600$ means 75% efficiency for "large" problem in contrast with 62.7% for "small" one.

6. Acknowledgements

This work was supported by Russian Foundation for Basic Research (grants No. 05-07-90230, 05-01-00510)

References

[1] E.V. Shilnikov and M.A. Shoomkov: Parallel Program Package for 3D Unsteady Flows Simulation. In G.R. Joubert et al. (Eds.): Parallel Computing. Advances and current Issues. Proceedings of international conference ParCo2001, pp. 238-245, Imperial College Press, London, UK. 2002.

[2] E.V. Shilnikov: Viscous gas flow simulation on nested grids using multiprocessor computer systems. In: Proceedings of Parallel Computational Fluid Dynamics Conference (Moscow, Russia, 2003), pp. 110-115, Elsevier Science BV. 2004.

[3] D.A. Lin and W. Rody: The flapping shear layer formed by flow separation from the forward corner of a square cylinder. J. Fluid Mech., vol. 267, pp. 353-376. 1994.

[4] D.A. Lin, S. Einav, W. Rody, J.-H. Park: A laser-Doppler velocimetry study of ensemble-averaged characteristics of the turbulent near wake of a square cylinder. J. Fluid Mech., vol. 304, pp. 285-319. 1995.

[5] G. Bosch, W.Rodi: Simulation of vortex shedding past a square cylinder with different turbulence models. Int. J. Numer. Meth. Fluids, vol. 28, pp. 601-616. 1998.

[6] B.N. Chetverushkin: On improvement of gas flow description via kinetically-consistent difference schemes, In B.N. Chetverushkin et al (Eds): Experimental modeling and computation in flow, turbulence and combustion, Wiley, Vol. 2, pp. 27-37. 1997.

[7] B. Perthame: The kinetic approach to the system of conservation laws. Recent advances in partial differential equations, Res. Appl. Math., Vol. 30, Masson, Paris. 1992.

[8] E. Oñate and M. Manzam: Stabilization techniques for finite element analysis for convective-diffusion problem. Publication CIMNE 183. 2000.

[9] S. Succi. The lattice Boltzmann equations for fluid dymanics and beyond. Oxford, Clarendon press. 2001.

[10] C. Hirsh: Numerical computation of internal and external flows, Vol. 1: Fundamentals of numerical discretization. John Wiley & Sons, A Wiley Interscience Publication. 1988.

Data Structures and Mesh Processing in Parallel CFD Project GIMM

B. N. Chetverushkin[a], V. A. Gasilov[a], S. V. Polyakov[a], M. V. Iakobovski[a], E. L. Kartasheva[a], A. S. Boldarev[a], A. S. Minkin[a]

[a]Institute for Mathematical Modelling, Russian Ac.Sci., 4-A, Miusskaya Sq., Moscow 125047

Key words: Computational fluid dynamics, three-dimensional evolutionary viscid flow, parallel algorithms, distributed computations.

The study is done under the auspices of Russian Ac. Sci. (The State contract No. 10002-251/OMH-03/026-023/240603-806) and RFBR (Project No. 04-01-08024-OFI-A).

1. Introduction

Rapid development of networks caused the growth of interest to heterogeneous systems applications in remote computations resulting thereby in the progress of special software. In this paper we present a CFD project GIMM carried out in the Institute for Mathematical Modeling, RAS. GIMM pursues the goal of joining up new results in numerical analysis with latest achievements in creation of network facilities. Special attention is paid to the reasonable management of multiprocessor systems (MPS) with distributed memory. Design of new parallel algorithms and proper programming is considered as having paramount importance in effective exploitation of parallel systems. Design of new numerical methods is implemented in close correlation with the development of algorithms for parallel computing. The use of high-dimension irregular meshes permits to approximate accurately realistic 3D geometry of the simulated objects. Processing of irregular data structures is a complicated and very time-consuming numerical work. New effective algorithms of domain decomposition over processors taking into account numerical expenses in each point are developed that provides the good load balancing and minimize data exchange.

2. General description

The current version of GIMM components includes:

- CAD compatible data structures,
- Surface/solid grid generation tools,
- Libraries of a numerical package kernel,
- Problem-oriented application software for PC and parallel systems,
- Server tools for data storage, visualization, pre- and postprocessor tools for PC and parallel systems,
- Client tools for data storage, visualization, pre- and postprocessor tools for PC,
- Server control tools for processing the user tasks by parallel system,
- Client interface for PC.

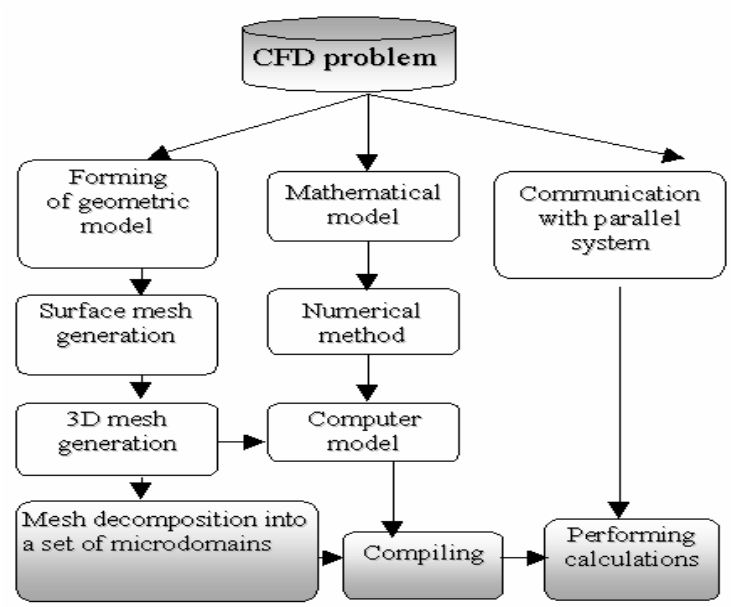


Figure 1. GIMM program manager.

The work of software components is supervised by the GIMM program manager. The first program manager version works under Windows and ensures the interactive user work with remote computing resources and computer systems possessing both distributed and shared memory. The server modules of the GIMM package are developed as Linux applications. The MPI is used for task start and interprocessor communications during the run. In case when the mixed type parallel systems are used for calculations, e. g. when the two- or four-processor nodes are incorporated into the computational net, the simulation can be implemented by means of both MPI and OpenMP utilities. To this end the work at the processors' level is done via algorithms using the common memory while at the nodes' level the appropriate algorithms use MPI utilities and distributed memory.

The program manager supports operations usual for CFD studies:

1. Preprocessing I, i. e. creation of a geometry model and setting of physical data as well as initial/boundary conditions.
2. Preprocessing II, or mesh generation and creation of a computational model equipped by problem attributes.
3. Formation of algebraic equations approximating the governing system of differential fluid mechanics equations.
4. Support of the task starting operations and run control functions.
5. Postprocessing, or data analysis and visualization in the remote operational mode. The program manager structure and main functions are shown on Fig. 1.

3. Numerical methods

At present in GIMM the following models are available:

- Model 1: 3D Navier-Stokes system for compressible heat-conductive flow;
- Model 2: 3D Navier-Stokes system for incompressible flow;
- Model 3: 3D single-phase nonlinear flow in porous media.

The governing systems are approximated by means of unstructured tetrahedral meshes and conjoint systems of finite volumes. The mixed finite volume (FV) — finite element (FE) approximations provide sufficiently high accuracy: the approximations to convective terms are done in terms of FV technique, and dissipative terms are approximated by simplex FE representations of dependent variables. For treatment of convective fluxes we use high-resolution TVD schemes of Roe, Osher and Van-Leer [1]. We also use the Chetverushkin kinetically-consistent method [2] and TVD Lax-Friedrichs schemes with intermediate solution reconstruction which we extended for the case of 3D unstructured meshes [3]. The time-marching algorithm is implemented as an explicit 2-nd order predictor-corrector.

3.1. Geometric models and meshing

Preprocessing related to geometric properties of a studied object usually is implemented by means of some CAD system. A set of universal program tools is developed for input and acquisition of geometrical and physical data related to a 3D problem with mixed initial and boundary conditions. Simultaneously these tools are used for storage and treatment of a discrete computational model. Our data model takes its origin in finite element technique and provides a description of geometric and topological properties for computational domains as well as for numerical meshes. The geometry of any computational model is formed of parametric surfaces expressed in the terms of rational B-splines (NURBS). Any 3D geometric complex is represented via sets of elements: 0-order elements = "nodes", 1-order elements = "edges", 2-order elements = "faces", 3-order elements = "cells".

A topology complex describes the relations between numbered data sets irrelative of their geometric nature. Typical 3D meshes consists of a large number of cells. Therefore only a few incidence relations are stored permanently and other relations are calculated every time they are needed.

The main topological relation between pairs of cells in a cellular complex is "to be a face". It is also named as a boundary relation. A cell d is a face of a cell e if d belongs to the closure of e. If $d \neq e$, then d is said to be a proper face of e. Other relations can be defined on the base of this one. Two cells e and d are incident if d is a proper face of e, or e is a proper face of d. Two cells e and d are adjacent if there exists another cell in the complex which is a proper face of both e and d. Thus the incidence relations serve for elements of different dimensions, and while adjacency relations serve for elements having the same dimension.

We suppose that computational mesh can change in the process of calculations. Data assigned to mesh elements are not predefined and can vary from one application to another so we introduced the conception of a numerated set. We use such sets for representation of collection of mesh elements. We imply that all mesh elements are enumerated and each of them has its unique number. So, having the number of an element, it is possible to find all the data assigned to this element. Numeration allows to implement changes in all data structures when some elements are added or deleted.

Evidently realistic results in 3D simulations can be achieved starting with meshes consisting of about $10^6 \div 10^7$ nodes or more. For such meshes the proper amount of a processed information is so large that it is necessary to develop parallel mesh generation algorithms. For applications in parallel computing we developed a technique based on domain decomposition with the following mesh generation by means of combined octree — modified advancing front algorithms [4] permitting to generate cells of prescribed shapes and sizes. The octree technique is applied for mesh generation

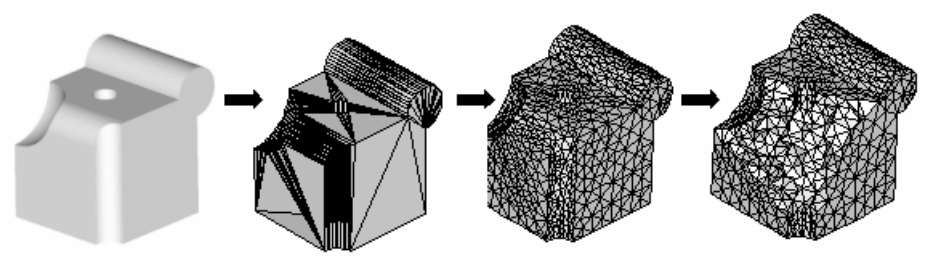


Figure 2. A scheme of mesh generation in a solid.

in the interior of any subdomain while the advancing front algorithm works in the near-boundary regions. Some optimizing procedures are developed for getting meshes of the higher quality. The initial front is formed by surface triangulation (see Fig. 2). The technique of multicolouring allows to implement parallel meshing in subdomains in such sequence that avoids the data exchange between adjacent regions and ensures fully independent meshing of subdomains.

3.2. The data treatment for the high-dimension meshes

Some problems are to be solved for the effective use of high-dimension meshes. The one is the correct load balancing whish is a very important factor strongly affecting a rate of parallel computations.

Another problem providing the evident difficulties in large-scale 3D simulations is the lack of acceptable visualization tools. For a typical mesh consisting of $10^6 \div 10^7$ nodes the whole amount of data exceeds the memory resources of a personal computer.

Let's consider the main aspects of computational technologies concerning the large volume data treatment created under the GIMM project.

The load balancing is done by means of the rational decomposition of the computational mesh via the constructing of a multilevel graph system. Due to this method mesh is represented as an ensemble of coherent subdomains. The initial graph is formed by pairs of neighbor mesh vertices. In turn, these pairs are combined into the higher level pairs. Thus the multilevel graph decomposition is formed successively, while every new graph possesses a simpler structure than the previous one. As the result of this work the computational mesh is represented as a set of connected subdomains. This procedure repeats until the graph of a proper size is obtained which can be easily divided into the required number of macrodomains. At the farther stage of the run these macrodomains are distributed over the set of processors. The macrodomains serve also for estimations of the numerical work. During the macrodomain creation performance it is possible to take into account not only the mesh topology, but an a priori estimated numerical complexity of calculations per every mesh node. Thus the resulted load distribution can be made very close to the optimum.

The data storage in the GIMM code is organized via the data distribution over the multiprocessor system itself. This technology is supported by specially developed hierarchical distributed file system (HDFS). It is based on the client-server technology in the multiprocessor variant. In accordance with this technology some number of specially appointed processors are used for communications through the entire disc space of the massive parallel system (MPS) including as local (processor self memory) as external (RAID server) devices. The other processors implementing the run are the clients connected to definite servers. The read/write operations corresponding to the distributed files

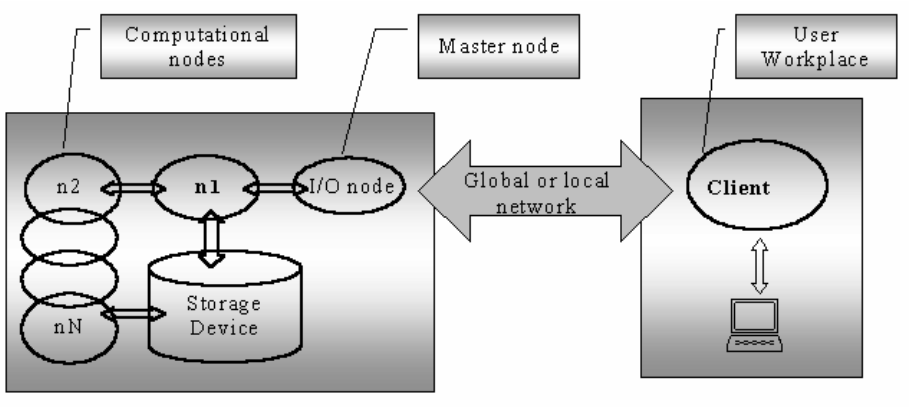


Figure 3. RemoteViewer: application of a client-server technology

are performed via the system of demands addressed to the HDFS servers. The cash memory using leads to somewhat information doubling which is almost inevitable. Therefore the main idea of organizing the HDFS system is to minimize the doubling of the information stored on the individual discs and simultaneously to maintain the high exchange rate and information integrity. In addition the HDFS is used for data compressing necessary for data storage and further treatment at the postprocessing stage. The convenient tools for data visualization and postprocessing are of primary importance in 3D simulations of complex FD problems. Here one should take into consideration the fact that typical volume of the 3D results obtained with a mesh of $\sim 10^6 - 10^7$ exceeds the storage capability of an individual PC. Besides, the data exchange through the net connecting PC with MPS is restricted by other operations related to multiprocessor computations.

To mitigate this problem we developed the distributed visualization system which is especially suitable for treatment of large data volumes. The GIMM visualization system RV — "Remote Viewer" is constructed according to a Client/Server model. It works in close interaction with HDFS. This construction allows to implement the most part of visualization process by supercomputer and then to transfer the compressed information to a user (see Fig. 3).

The final image is formed at the user's workplace and it is possible to use modern multimedia hardware (helmets, stereo glasses, three-dimensional manipulators etc.) for better image perception.

4. Numerical results

At modern stage of the GIMM development we performed hydrodynamic test studies pursuing the two main goals:

- approbation of the algorithms and numerical techniques (accuracy, efficiency, etc.);
- study of the effectiveness of parallel processing different distributed computing systems architecture.

The package working ability was examined through a number of test studies. Here we present a brief description of results pertinent to such famous benchmark as the problem of viscid low-Mach ($\mathrm{M} \sim 0.1$) and low-Reynolds ($\mathrm{Re} \sim 25$) flow around a sphere. A numerical grid was constructed so to condense in the sphere vicinity, and the volumetric ratio of largest cells to smallest ones was ~ 100. The total mesh consisted of 2 356 196 nodes and 14 018 176 tetrahedrons. Note that for

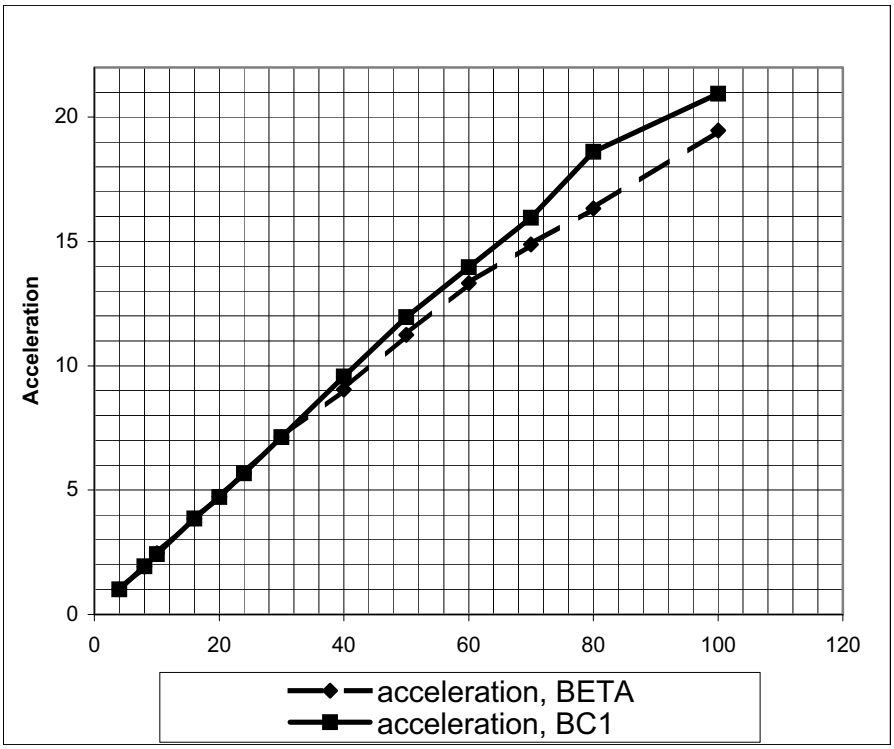


Figure 4. Acceleration rates for BETA and BC1 parallel systems.

the given mesh parameters the calculations with only a single-processor computer are almost out of perspective as we consider reasonable time of getting the result.

The steady-state calculations were done by the time-marshing procedure. For the test studies we used the two parallel systems operating in the Joint Supercomputer Center (Russian Ac. Sci.): MVS-1000 (equipped by Dec Alpha 21167 processors) and MVS-5000 (equipped by IBM Power P4 processors). The acceleration according to increased number of the used processors was estimated by the formula $S_N = T_4/T_N$, where N is the number of processors, T_N is the computing time for these processors. The effectiveness was estimated by the formula $E_N = (S_N \cdot 4/N) \cdot 100\% = (S_N/(N/4)) \cdot 100\%$. One may see from the figure 4 that maximum effectiveness for MVS-1000M was 98.3% when $N = 10$, and the minimum effectiveness was 77.8% when $N = 100$. These results demonstrate that the algorithm is efficient even when using strongly irregular tetrahedral meshes.

5. Conclusion

Dealing with the standard industrial prgram tools the user may encounter the restrictions of his activity caused to some extent by the stiff program structure or by the set of standard models and methods. Being reasonable in CAD-CAE project design, such restrictions fall into contradiction with the concept of a research code which is often aimed at studies of new models and algorithms. Considering the application aspects of high-performance calculations one may conclude that the industrial codes yet can not use perfectly all the resources which can be given by modern massive computational systems. Keeping in mind these points the GIMM team concentrated efforts on the development of open code with predominant applications in the area of parallel and distributive computations. The main goal pursued at the first stage of the project development was the develop-

ment of versatile algorithms suitable for various applications in CFD studies. This was the principal motivation for the development of numerical algorithms using unstructured meshes and their adaptation to calculations with various multiprocessor systems. The modern version of the GIMM code is aimed at numerical simulation of evolutionary multiscale 3D hydrodynamic phenomena. It combines as traditional as new numerical technologies which were incorporated into the code with the primary aim to have a universal simulation tool. GIMM provides a possibility of massive parallel computations with $10^8 \div 10^9$ mesh nodes and by the order of 10^3 processors involved. The main feature of GIMM package is the comprehensive use of parallel programming at all stages of a problem solution, i. e. from geometry modeling till postprocessing. Some numerical tools which are already incorporated into GIMM were developed 3–5 years ago and passed through comprehensive approbation in IMM RAS. The created code is highly effective compared to the traditional PC-oriented tools and allows 10 to 100 times reduction of the period necessary for the numerical study of as fundamental as applied problems.

References

[1] D. Kroner: Numerical schemes for conservation laws. B. G. Teubner Publ., Stutgart, Leipzig. 2000.

[2] B. N. Chetverushkin: Kinetic schemes and quasigasdynamic system. MAKS Press, Moscow. 2004.

[3] V. A. Gasilov and S. V. D'yachenko: Quasimonotonous 2D MHD scheme for unstructured meshes. Mathematical Modeling: modern methods and applications. Moscow, Janus-K. 2004. P. 108–125.3

[4] P. J. Frey and P. L. George: Mesh generation. Hermes Sci. Publ., Oxford, UK. 2000.

[5] M. Yakobovski, S. Boldyrev, and S. Sukov: Big Unstructured mesh Processing on Multiprocessor Computer Systems. In: Parallel Computational Fluid Dynamics - Advanced numerical methods, Software and Applications / B.Chetverushkin, A.Ecer, J.Periaux, M. Satofuka and P.Fox (Editors). Elsevier B. V. 2004. P. 73–80.

Radiative gas dynamics parallel computing using unstructured meshes

B.N.Chetverushkin[a], V.A.Gasilov[a], O.G.Olkhovskaya[a], S.V.D'yachenko[a], E.L.Kartashova[a], A.S.Boldarev[a], and V.V.Valko[b]

[a]Institute for Mathematical Modelling, Russian Ac.Sci., 4-A, Miusskaya Sq., 125047, Moscow, Russia

[b]The Central Physical-Technical Institute of the RF Defence Ministry, Sergiev Posad, 141300, Moscow region, Russia

The study is supported by the State contract 10002-251/OMH-03/026-023/240603-806, ISTC project 2830, and RFBR project 04-01-08024-OFI-A.

Key words: Computational fluid dynamics; radiative transfer, unstructured mesh; parallel algorithms;

High-performance computing evidently is a proper tool for simulations of transient high-temperature flows. A lot of numerical work is typically required to resolve accurately complex multiscale nonlinear processes in hypersonic gases and plasmas. Taking this as a starting motivation we develop a new object-oriented code for simulations of coupled radiative-gasdynamics processes using unstructured grid technology. The first applications of this new code showed promising results, which make us confident in good perspectives of using the unstructured grid technologies in simulations of complex physical models. The benchmark problem presented here is the computation of hypersonic radiating flowfield over a blunt body simulating the space vehicle re-entry conditions.

1. Mathematical models and numerical methods

The code performs calculations in planar or cylindrical coordinates. We accept a single-fluid gasdynamics model including dissipative processes [1]. The gasdynamics equations are written in the form of conservation laws.

- continuity equation $\dfrac{\partial \rho}{\partial t} + \operatorname{div}(\rho \mathbf{V}) = 0$

- momentum equation $\dfrac{\partial \rho \mathbf{V}}{\partial t} + \operatorname{div}(p\mathbf{V} \otimes \mathbf{V}) = -\operatorname{grad} P + \operatorname{div}(\hat{\tau})$

- energy balance equation $\dfrac{\partial (\rho\varepsilon)}{\partial t} + \operatorname{div}(\rho\varepsilon \mathbf{V}) = -P \operatorname{div} \mathbf{V} - \operatorname{div} \mathbf{Q} - G_{\mathrm{Rad}} + \Phi$

viscosity forces: $(\hat{\tau}) = \mu \Delta \mathbf{V} + \left(\xi + \dfrac{\mu}{3}\right) \operatorname{grad}(\operatorname{div} \mathbf{V})$

viscous dissipation of energy: $\Phi = \sum\limits_{i,k} \dfrac{\mu}{2} \left(\dfrac{\partial V_i}{\partial x_k} + \dfrac{\partial V_k}{\partial x_i} \right)^2 + \left(\xi - \dfrac{2}{3}\mu \right) (\operatorname{div} \mathbf{V})^2$

ξ, μ — viscosity coefficients

Common notations for physical values is used here. The full internal energy ε includes the energy of vibration excitement, dissociation and ionization. The last equation is taken in a one-temperature approximation and includes radiation losses. We suppose that the flow exists under LTE (local thermodynamic equilibrium) condition and energy consumption for vibration excitement, dissociation, and ionization is accounted by the use of tabulated internal energy dependence upon density and temperature.

The solution of the RGD system is done by use of unstructured triangular mesh. The main advantage of unstructured meshes is the ease of boundary fitting and refinement procedures implementation. We utilize the node-centered (nonstaggered) representation of calculated values. The technique of finite volumes is used for approximation of the governing system. In the present program version the finite volumes are formed by modified Voronoi diagrams. The triangular mesh in the computational domain and a zoomed fragments of the initial triangular grid and the finite volumes near the body head are shown below.

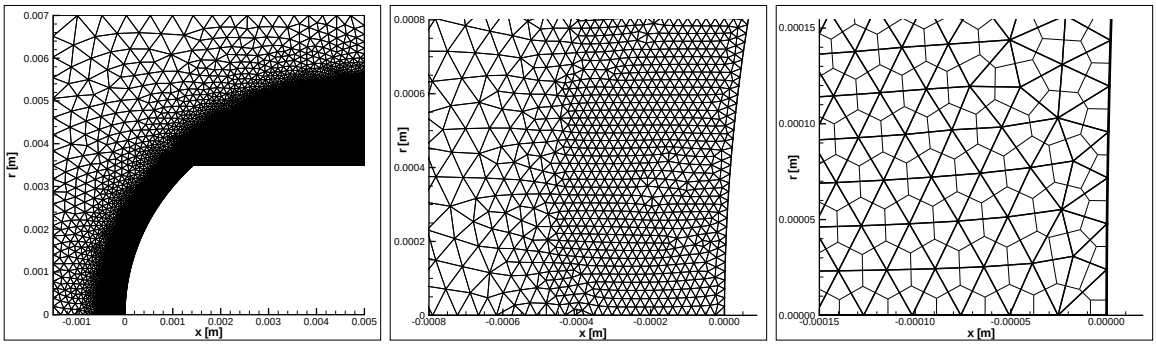


Figure 1. The triangular mesh in the computational domain.

A splitting scheme is applied to the governing system with the subsets of equations describing different physical processes being solved in sequence by appropriate program modules. The GD system is solved by the generalized TVD Lax-Friedrichs scheme which was developed especially for the unstructured mesh applications [4]. For the case of a regular triangulation this scheme ensures the second order approximation to spatial derivatives (the third order is possible with a special choice of the antidiffusion limiters). The time integration is explicit, the second approximation order is reached due to the predictor - corrector procedure. The time step is restricted by the Courant criterion. For the solution of parabolic equations describing the conductive heat transfer, we developed the new finite-volume schemes constructed by analogy with mixed finite-element method.

Radiative energy transfer is described by the equation for spectral radiation intensity. Practical calculations are done via multigroup spectral approximation. We solve the radiative transport equation by means of semi-analytical characteristic algorithm. The analytical solution along the characteristic direction is constructed by means of the backward-forward angular approximation to the photon distribution function [2], [3]. The two-group angular splitting gives an analytical expression for radiation intensity dependent on opacity and emissivity coefficients. The radiation-matter energy exchange is taken into account via a radiative flux divergence, which is incorporated into the energy balance as a source function.

Transport equation for quasistationary radiation field in cylindrical geometry:

$$\sin\theta\left(\cos\varphi\frac{\partial I_\omega}{\partial r}+\frac{\sin\varphi}{r}\frac{\partial I_\omega}{\partial\varphi}\right)+\cos\theta\frac{\partial I_\omega}{\partial z}=-\aleph_\omega I_\omega+j_\omega$$

A set of equations for the forward/backward intensity functions $I^{f/b}$:

$$\frac{\cos\theta_{n+1}-\cos\theta_n}{\Delta\theta_n}\left(\frac{\partial I^b_{n+1/2}}{\partial r}+\frac{I^b_{n+1/2}}{r}\right)+\frac{\sin\theta_{n+1}-\sin\theta_n}{\Delta\theta_n}\frac{\partial I^b_{n+1/2}}{\partial z}=-\aleph I^b_{n+1/2}+j$$

$$\frac{\cos\theta_n-\cos\theta_{n+1}}{\Delta\theta_n}\left(\frac{\partial I^f_{n+1/2}}{\partial r}+\frac{I^f_{n+1/2}}{r}\right)+\frac{\sin\theta_{n+1}-\sin\theta_n}{\Delta\theta_n}\frac{\partial I^f_{n+1/2}}{\partial z}=-\aleph I^f_{n+1/2}+j$$

The radiation energy density

$$U=\frac{\pi}{c}\sum_{n=1}^{N}\left(I^f_{n+1/2}+I^b_{n+1/2}\right)(\cos\theta_n-\cos\theta_{n+1})$$

The forward/backward intensities along a ray in the direction $\theta_{n+1/2}$

$$I^{f/b}=(I^{f/b}_{i,j+1}-I^{eq}_{i,j})\exp(-\kappa_{i,j}\xi_{i,j})+I^{eq}_{i,j}$$

For the purpose of the radiation energy transport calculation a special grid of characteristics is constructed in the computational area. This grid is formed by a number of sets (families) of parallel right lines and is further referred to as the grid of rays. Each set of parallel lines is characterized by the angle of inclination to coordinate axes and spatial density of rays. The grid of rays introduces some discretization of the computational area in the plane (r,z) and also with respect to the angle θ, $(0<\theta<\pi)$, which is required for numerical integration of the radiation transport equation according to the described above model. The grid of rays is superimposed on the initial computational grid intended for gas dynamics and heat transfer computations. A fragment of the grid of rays (12 angle sectors) is shown at the figure.

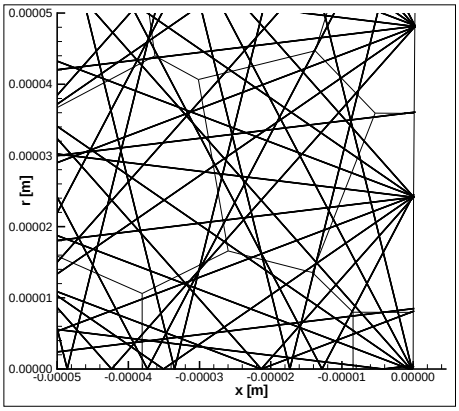


Figure 2. A fragment of the grid of rays.

2. Thermodynamic and optical properties of the air

For practical applications the datatables of material properties as to thermodynamics, ionisation, opacities and emissivities are used. A wide-range equation of state was constructed for the air composition: $78.12\% - N_2$, $20.95\% - O_2$, $0.93\% - Ar$ (hereinafter - the air). The effective ranges of the thermodynamic functions values in the developed equation of state are the following:

for the temperature: 200 to $1.16 \cdot 10^5$ K;

for the density: 10^{-3} to 18 g/m^3;

for the pressure: $5.7 \cdot 10^{-8}$ to 3.7 GPa;

for the specific internal energy: $1.4 \cdot 10^{-1}$ to $8.3 \cdot 10^4$ kJ/g.

The model of plasma imperfection with the classic Coulomb correction Debye-Hukkel in big canonical ensemble (BDH) was applied in the computations. Generation and dissociation of two- and three- atom molecules were accounted as well as adhesion of electrons at the atoms and molecules. Statistical sums for the atoms and molecules were calculated using up-to-date statistical sums truncation models. Up to 10-15 electron states were considered in calculation of the full statistical sums. Microfield truncation formfactor was employed for statistical sums calculation for atoms and ions. The molecules N_2, O_2, NO, their first positive ions, negative ions O_2, NO, O, and all the atomic positive ions up to their maximum charge number were considered in the computations. (See the figure a.)

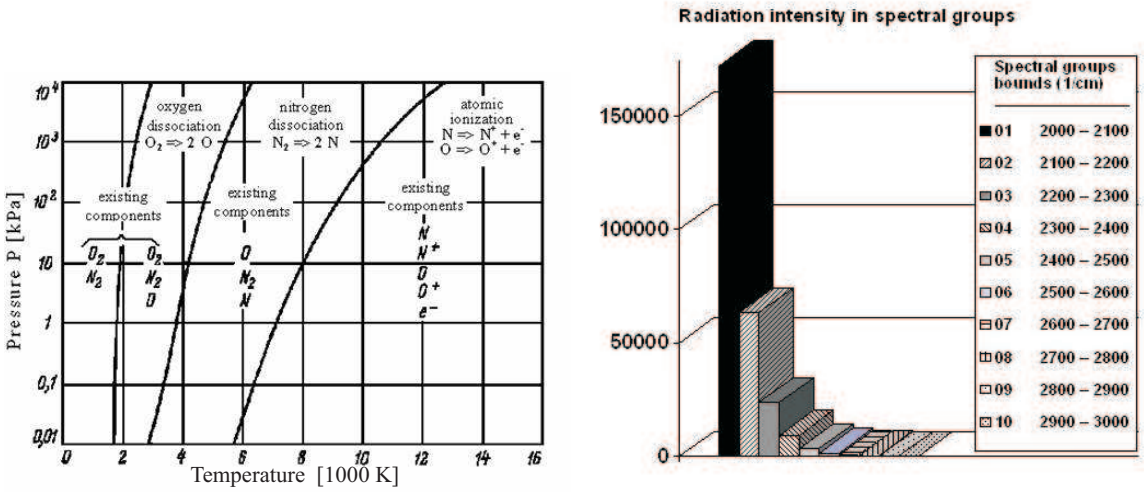


Figure 3. a) Dissociation bounds of the air components. b) Radiation in spectral intervals.

The spectral absorption coefficients were calculated for 20 spectral groups (Plank and Rosseland averaging as well as that with a unit weight factor) in the range of temperature T from 200 to $5.5 \cdot 10^4$ K and the range of relative density ρ from 10^{-3} to 10. The opacities were calculated for the wavenumber band from 2000 cm^{-1} to 500000 cm^{-1} split uniformly by 250 groups. Due to the required temperature range the mathematical modelling of the air composition optical properties was accomplished here by the use of semiempirical model of continuous spectrum. The calculations were done taking into account the influence of spectral lines. Then 20 group spectral model of the high-temperature air composition opacity coefficients was constructed for the radiation gasdynamics

tasks, the boundaries of the averaging ranges being chosen purposely to provide the best description of the radiation energy transport for T near $20 \cdot 10^3$ K. The Plank and Rosseland averaging as well as that with a unit weight factor was carried out. Noteworthy that the maximal deviation in integral Plank absorbtion coefficient between our calculated value and those obtained by another technique independently was only 4%.

First the 20 spectral groups bounds were chosen in the range 2000 - 500000 cm^{-1}. But the numerical experiments showed that the entire radiation is concentrated in the first group (2000 - 4000 cm^{-1}). Then a new set of spectral groups was introduced to be used in practical computations. The above diagram (the figure b) demonstrates the radiation intensity distribution in the first 10 of these groups. The radiation in the rest 10 groups is almost negligible. Radiative energy flux was evaluated at the body head surface.

3. Data structures and parallel implementation

A universal program tools were developed convenient for input and acquisition of geometrical/physical data as well as for storage and treatment of a discrete computational model. Cellular model is used for both the computational domain and the meshes described as geometric complexes. A formalized description of geometric and topological properties of meshes is thus provided. Topological complexes of various dimensionality are suitable for representation of irregular continuum as well as discrete structures. 3D geometric complex includes: 0-order elements (nodes), 1-order elements (edges), 2-order elements (faces), and 3-order elements (cells). The boundary of each element is formed by the elements of lower order. Topology is described by relations between the elements. Two types of relations are used: incidence relations (between elements of different dimensions) and adjacency relations (between elements of the same dimension). Only a few base incidence relations are stored permanently and other relations are calculated every time we need them. Dynamic topology changes are supported. Each element is identified by its number. Having the number of an element, it is possible to find all the data assigned to this element. Special methods were developed that allow to implement for all data structures appropriate changes caused by variations in elements numeration (e.g. adding or removing the elements while grid construction or refinement).

This geometric data treatment technique is especially effective for unstructured grids, but may be applied for regular grids as well. It is also useful for handling subdomains processed by a distributed computer system.

The developed explicit difference scheme allows quite natural parallel implementation for distributed computer systems. Each processing node is associated with a section of the triangular computational grid (a subdomain). Each processor carries out the computations only inside this subdomain. Subdomains cover the entire computational grid and may have common nodes only at the boundaries. The data exchange between subdomains is organized through the "margins". We called so the layers of grid elements belonging to the neighboring subdomains. In that way some elements of a subdomain are included in the "margins" of the neighboring subdomains. Any data from a margin node are to be retrieved via high-speed network from the processor where this node is stored as an element of the related subdomain.

Numerical experiments showed that it is necessary to pay a special attention to the proper domain decomposition in order to minimize the problem time. The load of the processors should be well balanced. Otherwise downtime is inevitable when an underloaded processor finishes a step of calculations and is waiting until the neighboring overloaded processors will be able to provide the requested data. We use MPI for distributed computations and ParMetis for mesh partition.

Another important factor is the avoidance of collisions, i.e. the proper time balancing of the net-

work channel loading. The processors should not request the network data exchange simultaneously with the network being then idle for a relative long time. A downtime may occur because of insufficient network throughput thus increasing the overall computation time notably. This problem is fixed by a special order of calculations in the internal and boundary nodes in subdomains.

Implementation of complex physical models incorporating multi-scaled and essentially non-local processes requires more than one grid structure. For instance, an additional grid of rays (characteristics) is necessary for the simulation of radiative energy transfer by the method of characteristics. As a rule, the number of elements (and therefore the resources consumption) of these grids are comparable. In this case it is reasonable to arrange distributed computing involving not only domain decomposition but the splitting of the physical processes as well. That means that a special group of processors may be provided for the radiation transfer simulation. The multigroup model allows separated computations for each spectral group. Thus the spectral groups may be distributed for a several processors. It is important that an intensive non-local data exchange exists between the different grid structures. When the transport equation is solved along the characteristics each of them crosses several subdomains and both the internal and boundary cells are crossed. Thus additional conditions are introduced into the interprocessor communications balancing problem - we have to optimize the rays distribution over the triangular grid partition. Rays crossing the basic grid subdomains are schematically shown at the diagram.

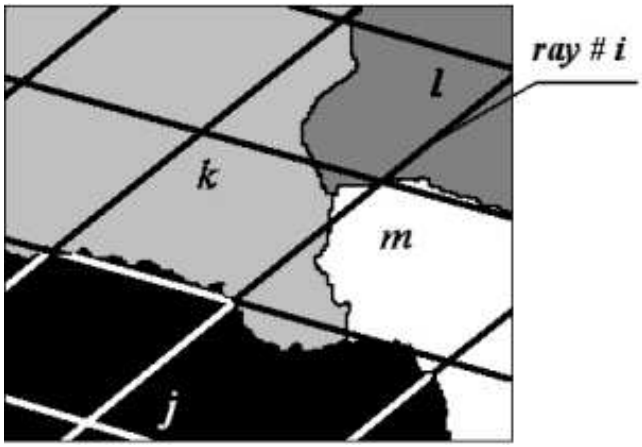


Figure 4. The grid of rays over domain partition.

The basic triangular grid is partitioned to N domains located at N processors ($N = 16$ in our case). We are to distribute M rays (grid of characteristics) over the same processors ($M = 3845$). Each ray is divided into segments, associated with the basic grid cells (control volumes). Each ray is processed individually. The computations are carried out along a ray, and each segment utilizes the calculated data from the related cell (because the optical properties at a segment are dependent upon the temperature and the density in the cell). That means that if the ray $\#i$ is processed by the processor j and the cells involved are located in the domains j, k, l, m, only the data from the domain j cells may be acquired directly, and those from the domains k, l, m require communications between the processors $j-k$, $j-l$, $j-m$. The number of ray segments varies significantly. Some rays may include only two segments and some a hundred or more. A lot of rays cross several domains. That's why an optimization of ray distribution is important.

Boolean linear programming problem statement.

Unknown variables: $x_{ij} = \begin{cases} 1 & \text{if the ray } \#i \text{ is at the processor } j \\ 0 & \text{otherwise} \end{cases}$

Limitations: each ray should be included in the partition exactly once: $\forall i \sum\limits_{i} x_{ij} = 1$.

Criterion function 1. The number of interprocessor communications.

$$F_1 = \sum_i \sum_j a_{ij}(1-x_{ij}) \to min$$

Weight factor a_{ij} is the number of ray $\#i$ segments located in the domain j. It is the number of segments acquiring the information from the processor j. In particular $a_{ij} = 0$ if the ray $\#i$ does not cross the domain j.

Criterion function 2. The processor load balance.

$$F_2 = \sum_k \sum_l (\varphi_k - \varphi_l)^2 \to min$$

Here $\varphi_j = \sum\limits_i b_i x_{ij}$ is the number of rays segments processed by the processor j.

$b_i = \sum\limits_j a_{ij}$ is the total number of ray $\#i$ segments.

The aggregate criterion function: $F = \alpha F_1 + \beta F_2$.

The factors α and β depend upon the estimations of the interprocessor communication time (α) and the ray processing time (β).

For practical computations a heuristic optimization algorithm was applied.

4. Numerical results

The constructed algorithms were examined in numerical experiments related to the problem of satellite re-entry studies.

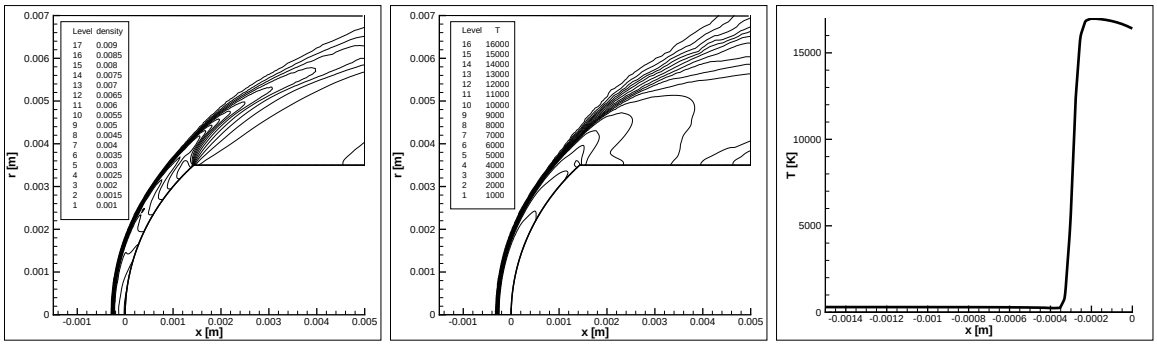


Figure 5. Calculated density and temperature distribution.

To verify the accuracy of the presented approaches the calculations was made for the flow around the spherically nosed cylinder with 0.5 cm radius and 0.7 cm cylinder diameter. In the present study the shock layer flow over this model at the free-stream density $5.5 \cdot 10^{-4}$ kg/m^3 and the temperature

300K was studied for two different values of the free-stream velocity 13.4 km/s and 16 km/s. We suppose the following boundary conditions: a slip conditions at the wall, a zeroth-order extrapolation at the outflow boundary, a symmetry condition in the symmetry plane, and the free-stream condition at the inflow boundary. For the radiation processes the far-field is assumed be a black body having a temperature 300 K. The wall is assumed to be a black body at 3000 K.

The simulations showed an excellent resolution of flow structures and high performance capability of the developed code. The above figures illustrate density and temperature distribution near the blunt body and the temperature distribution along the stagnation line for the case of the free-stream velocity of 16 km/s.

The numerical results are in good agreement with experimental data given in [5].

5. Conclusions

1. The numerical simulation is an important tool for re-entry studies as it allows to circumvent the problem of absence of "scaling" parameters at presence of chemical transformations, which complicates theoretical studies of hypersonic flows.

2. The flowfield around a re-entry satellite can be simulated quite well by means of numerical technology based on unstructured grids locally refined for precise treatment of shocks and boundary layers. The corresponding results are in good agreement with experimental observations as well as theoretical predictions and previous numerical studies.

3. For the studied problems the radiative transfer exerts rather small influence on the whole energy balance in the near-body region. Taking into account the wall evaporation by means of introducing some amounts of gaseous carbon into the boundary layer has no important influence on the radiative properties of the flow. The energy balance under the assumed flight conditions depends mainly on dissociation-recombination in the air past the head shock wave and in the boundary layer.

4. Further investigations of the re-entry problem require taking into consideration the nonequilibrium chemical transformations in the shock.

5. To estimate more comprehensively the influence of radiative transfer it is necessary to simulate the re-entry flow for various flight conditions, i.e. for various altitudes, free-stream Mach number and at various attack angles.

6. For the last item the application of 3D simulation tools is necessary.

References

[1] L.D.Landau and E.M.Livshits. A course in theoretical physics. Vol VI "Hydrodynamics". Moscow, "Fizmatlit", 1986.

[2] R.Siegel, J. R.Howell. Thermal radiation heat transfer. Tokyo, McGraw-Hill, 1972.

[3] B.N.Chtverushkin. Mathematical modeling in radiative gasdymnamic problems. Moscow, Nauka publ., 1985.

[4] V.A.Gasilov and S.V.D'yachenko. Quasimonotonous 2D MHD scheme for unstructured meshes. Mathematical Modeling: modern methods and applications. Moscow, Janus-K, 2004, pp.108-125.

[5] Sakai T., Tsuru T. and Sawada K., Computation of Hypersonic Radiating Flowfield over a Blunt Body, Journal of Thermophysics and Heat Transfer, Vol. 15, N 1, 91-98, 2001.

Performance analysis and visualization of the N-body tree code PEPC on massively parallel computers

P. Gibbon[a], W. Frings[a], S. Dominiczak[a], B. Mohr[a],

[a]John-von-Neumann Institute for Computing, Forschungszentrum Jülich GmbH, ZAM, D–52425 Jülich, Germany

The performance and scalability of a parallel tree code for rapid computation of long-range Coulomb forces is investigated using both visual and analytical techniques. The present code uses a variation of the Hashed-Oct-Tree algorithm, in which communication overhead is minimised by bundling multipole data for large groups of particles prior to shipment between processors. The two critical components of this algorithm, the tree traversal and load-balancing, are examined in highly dynamic physical context with the help of the KOJAK performance analysis toolkit and the online visualisation packages VISIT and XNBODY. The parallel scalability of PEPC is investigated on the Jülich IBM p690 and BlueGene/L machines.

1. Introduction

Even in the era of Teraflop computing, the N-body problem for systems dominated by long-range potentials remains a formidable algorithmic and computational challenge. The brute-force approach, in which all $N(N-1)$ mutual interactions between simulation particles are computed directly, is simply impractical for many N-body systems such as plasmas, gravitational systems, or large molecules in ionized solution. This is particularly true when the global dynamic behaviour of the system is of primary interest, rather than the microscopic details of individual particle trajectories. For this class of problem there is often no need to compute potentials and forces to higher accuracy than the error incurred in integrating the equations of motion, typically in the 10^{-4}–10^{-2} range.

Two techniques developed in the mid-1980s—the hierarchical Tree Code [1] and the Fast Multipole Method (FMM) [2], with respective algorithmic scalings of $O(N \log N)$ and $O(N)$—have revolutionized long-range N-body simulation across a broad range of fields [3]. These methods reduce the number of direct particle-particle interactions through the systematic use of multipole expansions, making it possible perform simulations with many millions of particles. Despite this progress on the algorithmic side, recent advances in the massively parallel computing paradigm have prompted a further challenge: can hierarchical algorithms be effectively implemented on a parallel machine with thousands of processors?

At first sight, the recursive data structure of tree codes would seem to rule out parallelism altogether, but in fact the construction of both the tree and particle interaction lists can be cast in data-parallel form on a shared-memory machine [4], leaving a straightforward $N \times N_{list}$ force summation to contend with. On a distributed-memory machine, the tree structure either has to be known to all processors—restricting the maximum simulation size—or somehow divided up equally among them. In the latter case, a *locally essential* tree can be built comprising only the information required to compute forces for locally held particles. Over the past decade various parallel tree algorithms have been proposed and implemented, including virtual shared-memory approaches [5], and distributed memory schemes [6,7].

This paper describes an efficient, portable implementation of a parallel tree code—PEPC (Pretty Efficient Parallel Coulomb-solver)—initially designed for mesh-free modelling of nonlinear, com-

plex plasma systems [8], but recently extended to other application areas ranging from molecular dynamics to protoplanetary accretion discs [9]. For PEPC, we have adopted the Warren-Salmon 'hashed oct-tree' scheme based on a space-filling 'Morton' curve, derived from 64-bit particle-coordinate keys. The discontinuities inherent in this curve, potentially leading to disjointed domains and additional communication overhead [10] is found to be a relatively minor issue compared to load-balancing and geometrical factors.

While the performance of PEPC on both commodity and high-end clusters is such that multi-million-body simulations can already be routinely performed, porting the code to new architectures with many *thousands* of processors such as BlueGene/L presents a much tougher challenge. To isolate and unravel the communication-critical parts of the code more systematically, we have made use of the automatic performance toolkit KOJAK [11].

PEPC has also been equipped with a combination of visualization toolkits VISIT [12] and XNBODY [13] to assist in tracking progress and enable real-time computational steering (user-feedback) of simulations. We have extended this online visualisation and steering (OVS) capability for PEPC by incorporating details of the tree structure on each processor, thus allowing visual monitoring of the inner, dynamic workings of the algorithm, such as the domain decomposition or load balancing.

The structure of this paper is as follows: In Section 2 we briefly review the hashed oct-tree algorithm and the various implementations of the tree-traversal routine available in PEPC. In Section 3 some benchmarks for 'static' systems are presented demonstrating the code's scalability on the Jülich IBM p690 cluster JUMP. The tree-traversal variations are then examined with the help of the KOJAK toolkit with a view to porting the code onto the BlueGene system. Finally, in Section 4 the dynamic evolution of the code performance for a plasma physics application is then discussed with the help of online visualization techniques.

2. Variations on the Hashed Oct Tree algorithm: asynchronous *vs.* collective

The Hashed Oct Tree algorithm is well documented in the literature [6] so we need not dwell on the details of tree construction here: features particular to the code PEPC are also described elsewhere [14]. A summary of the algorithm implemented in PEPC is depicted in Table 1, along with the theoretical scaling and relative effort for each major routine. All of the above routines can be performed in parallel, requiring an effort $O(N/P)$, give or take a slowly varying logarithmic factor.

Code region	Scaling	% CPU time
Domain decomposition: weighted key-sort	N/P	3
Construct local trees and multipole moments	$P \log N/P$	4
Construct interaction lists (tree walk)	$N/P \log N$	43
Compute forces and potential	$N/P \log N$	49
Update particle velocities and positions	N/P	1

Table 1
Algorithmic scaling and relative computational effort of major routines in PEPC. The symbols N and P represent the total number of particles and processors respectively.

As mentioned above, the HOT algorithm employs 64-bit keys derived from the (3-dimensional) particle-coordinates. Domain decomposition is acheived by cutting out equal portions of the sorted

particle key-list and allocating these to the processors. The fully parallel sort currently implemented is an adaptation of the PSRS (parallel sort by regular sampling) scheme originally proposed in Ref. [15]. Since the distribution of keys depends sensitively on the geometry of the system simulated—that is, whether the particles are initially arranged in a cube, sphere or more complex geometry—regular sampling tends to produce highly imbalanced particle numbers across the processors. To compensate this effect, we instead use load-weighted sampling, which allows for the actual distribution of keys along the whole space-filling curve. Problems may arise here if the key distribution is not finely enough resolved, a feature which we return to in Sec. 4.

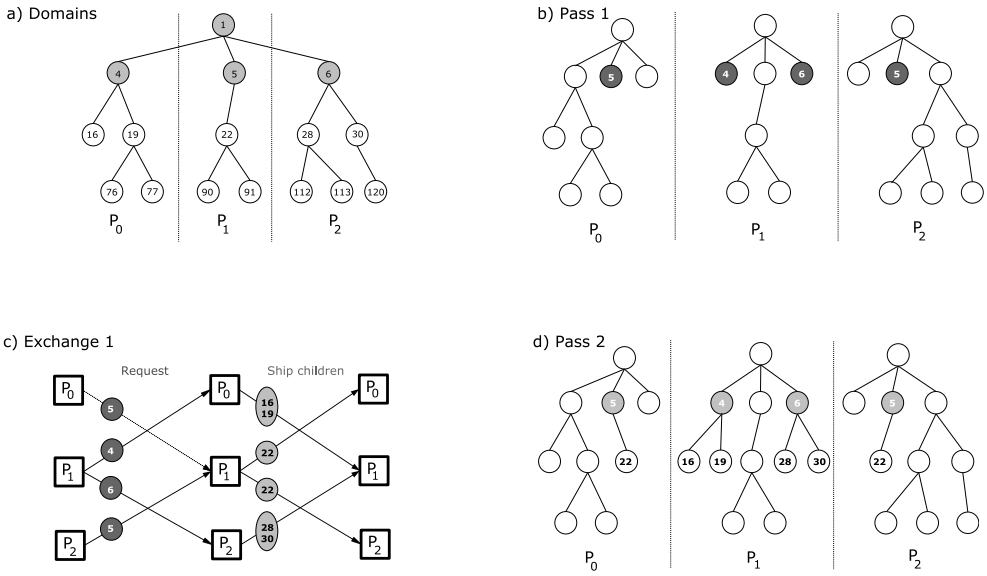


Figure 1. a) Domain decomposition and b)-d) parallel tree-walk for 3 CPUs.

To facilitate the exchange of information (in particular, multipole moments) between processors, a set of local 'branch' nodes is defined, comprising the minimum number of *complete* twig and leaf nodes covering the whole local domain—Fig. 2a). This set of branch nodes is then broadcast to all other processors, so that each one subsequently knows where to find (or request) any missing non-local particle or tree node.

By far the most algorithmically demanding part of this code is the tree walk, which in PEPC combines a previous list-based vectorised algorithm [4] with the asynchronous scheme of Warren & Salmon [6] for requesting multipole information on-the-fly from non-local processor domains. In the present scheme, rather than performing complete traversals for one particle at a time, as many 'simultaneous' traversals are made as possible, thus i) minimizing the duplication incurred when the same non-local multipole node is requested many times and ii) maximising the communication bandwidth by accumulating large numbers of nodes before shipment. In practice, this means creating interaction lists for batches of around 1000 particles at a time before actually computing their forces.

In the first pass of the walk (Fig. 2b), traversals are made through the local trees using the familiar divide-and-conquer strategy common to sequential tree codes [4]. The multipole acceptance criterion (MAC) determines whether to accept or subdivide local nodes as usual, but also provides for a third possibility: the subdivision of a *non*-local node for which child data is not yet available. This

is then placed on a special 'request' list (dark-shaded nodes) to be processed in the 2nd 'exchange' half of the routine (Fig. 2c) when all particles have completed their traversals as far as they can with the available node data. Each processor then compiles a lists of nodes it needs child data from, and sends them to the owners of the parent nodes. In the first pass, these will just be the branch nodes. On receipt of a request list, a processor packages and ships back the multipole data for the children. The use of non-blocking SENDS and RECEIVES for the multipole information in principle allows some overlap of communication with the creation of new local tree nodes (copies). At the end of subsequent passes (Fig.2d: here just 2 are needed), each processor's local tree contains all the nodes required to compute the forces on its own particles. With increasing system size, the nodes fetched during the traversals eventually take up most of the space in the local hash-table.

Three further variations of this procedure are currently implemented in PEPC: i) a *prefetch* mode in which lists of the fetched and requested nodes are retained for the subsequent timestep, allowing most of the locally essential tree to be rebuilt via a prune-and-graft procedure; ii) a purely collective exchange replacing the asynchronous SEND/RECEIVE swaps for each pass and iii) a *freeze* mode in which the entire tree structure is held fixed for several timesteps, but where the multipole information is updated and exchanged where necessary.

Once an interaction list has been found for a particle, it is a straightforward task to compute its force and/or potential. Separation of the actual force sum from the tree traversal has the advantage that this floating-point-intensive routine can be cache-optimised. Also, the physics and algorithm are kept naturally apart, so that additional forces and/or boundary conditions can be added with relative ease. In the present implementation, forces are computed for each batch of interaction lists returned from the tree-walk routine. One subtlety which arises here is that even if overall load-balancing has been arranged during the domain decomposition, it is not necessarily guaranteed for each batch of particles. To redress this problem, the batch size N_b for each processor is determined individually, so that the integral $\sum_{p=1}^{N_b} N_{\mathrm{int}}(p)$ is the same, and each processor computes the same number of interaction pairs during each pass.

3. Analysis of algorithm performance and scaling using KOJAK

The KOJAK performance-analysis tool environment [11] provides a complete tracing-based solution for automatic performance analysis of MPI, OpenMP, or hybrid applications running on parallel computers. KOJAK automatically searches execution traces of the application for patterns that indicate inefficient use of the underlying programming model(s). The KOJAK analysis process is composed of two parts: a semi-automatic multi-level instrumentation of the user application followed by an automatic analysis of the generated performance data.

The instrumentation inserts extra measurement code to capture begin and end of important phases in the user code (e.g., subroutines or loops), MPI message transfers and collective operations, as well as OpenMP constructs. KOJAK can handle C, C++, and Fortran source code. If necessary, the application can also be linked to the PAPI library [16] for collection of hardware counter metrics as part of the trace file.

Running the instrumented executable generates a trace file in the EPILOG format. After program termination, the trace file is fed into the EXPERT analyzer. EXPERT transforms event traces into a compact representation of performance behavior, which is essentially a mapping of tuples (performance problem, call path, location) onto the time spent on a particular performance problem while the program was executing in a particular call path at a particular location. There are two classes of search patterns, those that collect simple profiling information, such as communication or execution time, and those that identify complex inefficiency situations, such as a receiver waiting for

the wrong message. The former are usually described by pairs of enter and exit events, whereas the latter are described by more complex compound events usually involving more than two events covering multiple locations, a situation which can easily arise in a tree code.

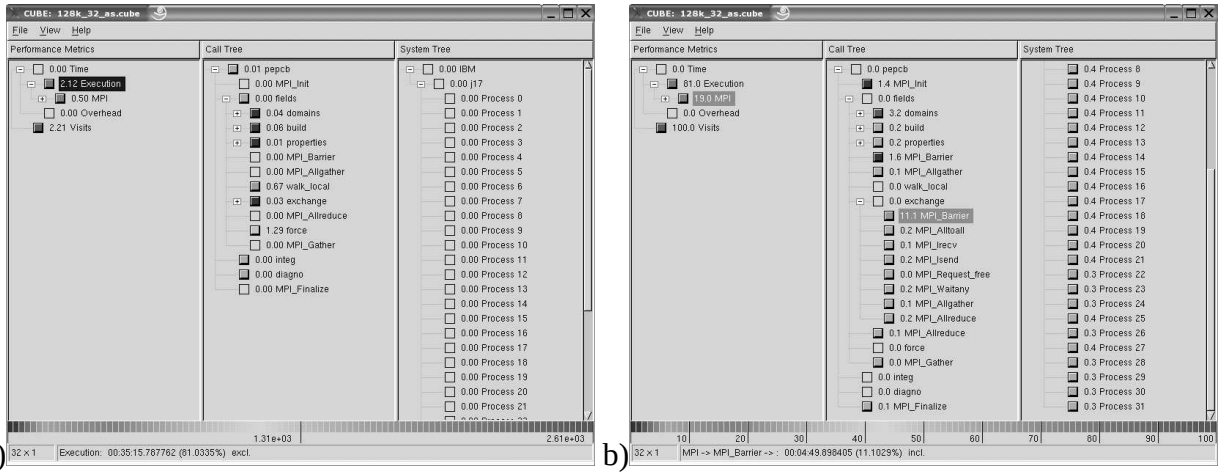


Figure 2. KOJAK EXPERT analysis of a 40-timestep PEPC test run using the asynchronous tree walk algorithm of Fig. 2 showing breakdown in: a) execution time and b) MPI-call time.

Figure 2 shows a screen-dump of the result presentation component (CUBE) of the EXPERT automatic event trace analyzer for a PEPC simulation of a sphere comprising 128000 charges. Using the color scale shown on the bottom, the severity of performance problems found (left pane) and their distribution over the program's call tree (middle pane) and machine locations (right pane) is displayed. The severity is expressed in percentage of execution time lost due to this problem. By expanding or collapsing nodes in each of the three trees, the analysis can be performed on different levels of granularity. We refer to [11] for a detailed description of KOJAK and EXPERT.

In this example the run took 43 minutes (2600 s) on aggregate (or 80 s per CPU): in the left-hand column we see how this is divided up in 'useful' execution- (81%) and MPI- (19%) time. Clicking on the Execution button we obtain the breakdown by routine in the 2nd column of Fig. 2a), where we find values consistent with those given in Table 1. The MPI breakdown in Fig. 2b) reveals a large imbalance, which is almost entirely due to nonlocal multipole fetches in the 'exchange' routine.

KOJAK also supports analysis of the *difference* between two measurements (e.g., using different input data sets, executing on different processor numbers, or comparing diferent implementations of the same code) by providing an utility that "subtracts" one CUBE result file from another one, resulting in an CUBE file which contains the differences of the severity for each problem for each call path on each loaction. This file can also be analyzed using the CUBE result browser.

This feature is exploited to compare the asynchronous tree-walk against the 'frozen tree' variation described previously for the same test problem as before – Fig. 3. Negative numbers (aggregate runtime seconds scaled according to bottom ruler) indicate an improvement over the asynchronous algorithm; positive numbers a deterioration. For the execution time, we see that there are substantial savings in the tree-building overhead (domains, build), which is what we expect by freezing the data structure and rebuilding only every 10th timestep. More significantly, communication time is also saved on the exchange part of the tree-walk, which is just what was intended. This is paid for by

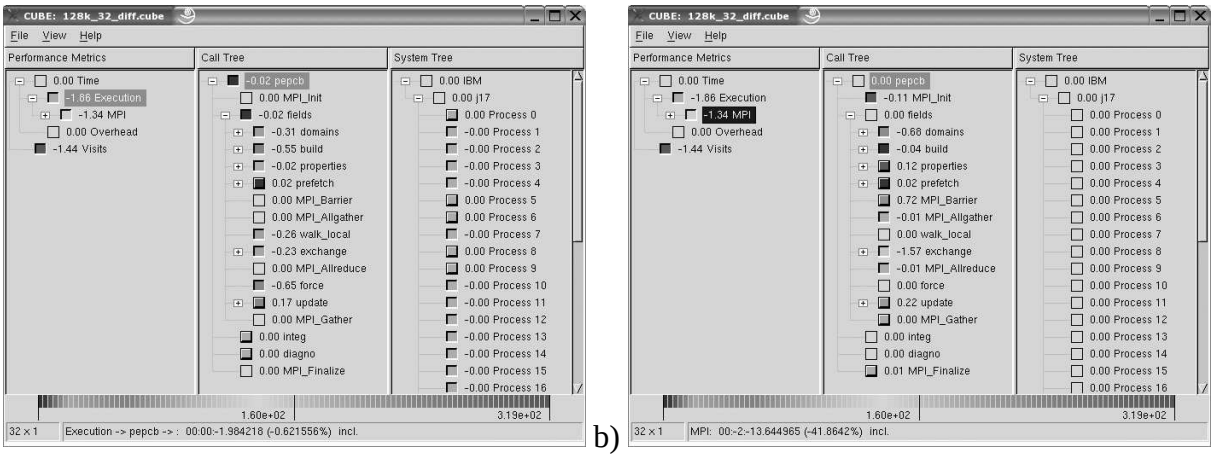


Figure 3. Differential analysis of PEPC sphere simulations comparing the asynchronous tree walk in Fig. 2 to the 'freeze' mode described in Sec. 2): a) execution breakdown; b) MPI breakdown.

the update routine, and, somewhat mysteriously, by increased barrier time – a side-effect which is presently not fully understood.

Overall however, the 'tree-freezing' concept shows promising scalability improvements over the asynchronous mode, as single-timestep benchmarks in Fig. 4 demonstrate. For large systems, the standard algorithm performs well on both the Regatta and BlueGene/L systems, scaling up to 1024 CPUs on the latter.

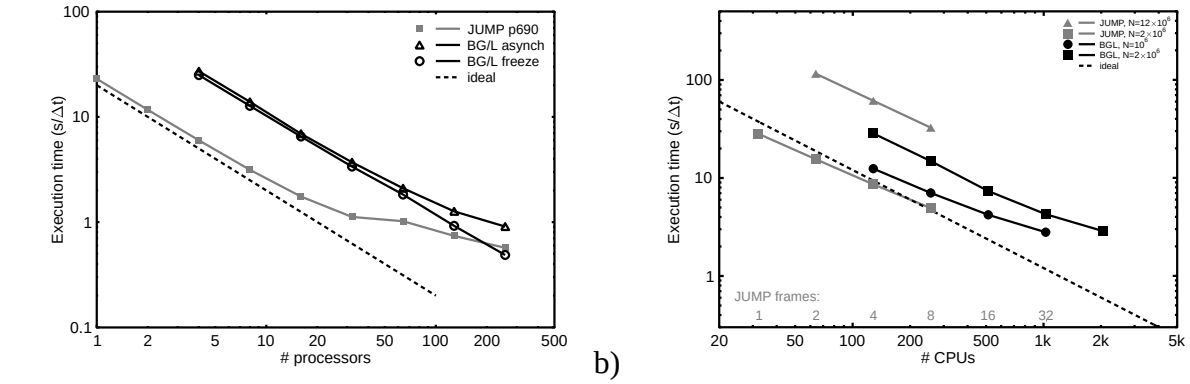


Figure 4. Timings on IBM-p690 cluster and BG/L for a) 128k and b) multi-million charge spheres.

4. Performance visualization using VISIT and XNBODY

Online visualization and steering (OVS) is normally used to track and adjust the dynamic development of simulations where the outcome depends on a large set of parameters. Here we use the OVS system developed at ZAM [13] to provide insight into the *algorithmic* behaviour of PEPC, and in particular to investigate the load-balancing characteristics for a dynamically evolving problem.

As noted before, this can be adjusted on the fly during the domain decomposition by appropriate weighting of the particle key-list segments allocated to each CPU.

To illustrate this in action, we consider an example taken from Ref. [8], in which a thin ionized wire comprising 1 million electrons and ions is irradiated by a high-intensity laser pulse. Physically, what happens here is that the laser begins to strip electrons from the target surface, accelerating them in all directions to form a rapidly expanding, negatively charged plume around the wire – Fig. 5(a–c). Despite the initially uniform density distribution, the target's geometry already poses problems for an unbalanced parallel tree code, as illustrated in the middle sequence (d–f), in which the particles are equally divided amoung the processors. The boxes represent local tree domains coloured according to the total amount of work performed by each CPU in the force calculation, which here varies by as much as 30%.

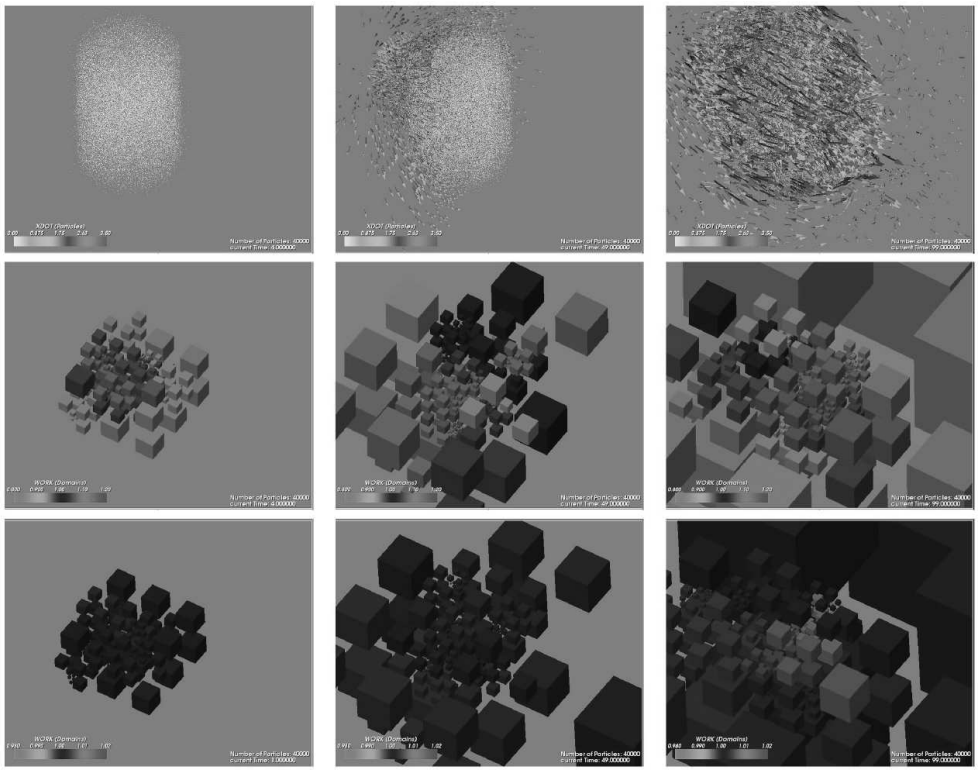


Figure 5. Visualization of dyanmic load balancing using VISIT and XNBODY toolkits. The top row shows the particle positions at times $t = 0, 50$ and 100 during the formation of the laser-induced charge cloud. The centre and bottom rows show tree domains, coloured according to the number of force computations per CPU for equal numbers of particles (middle) and load-balanced (bottom).

In the final sequence (g–i), the simulation has been repeated with dynamic load balancing switched on. In this case the imbalance is stays below 1% for most of the simulation. Towards the end however, we see that the workload apparently becomes uneven again near the centre of the target. Because of the expanding electron cloud, the system effectively becomes more clustered with time, leaving a high concentration of particle keys near the centre. This eventually leads to undersampling in the sort routine in this region, which in turn causes incorrect balancing — a feature which would

be difficult to trace without the direct visual relationship between work load and spatial location provided by this technique.

Summary

A new parallel tree code – PEPC – for rapid computation of long-range interactions has been presented in which various implementations of the tree traversal routine have been compared and the overall scaling of the code with up to 1024 processors of BlueGene/L demonstrated. Load balancing issues have also been investigated with the help of visual techniques, enabling potential pitfalls in the parallel sort routine to be properly addressed. As a result of these algorithmic improvements we expect PEPC to scale well beyond a single BG/L rack in the near future, paving the way for Coulomb/Newtonian gravity simulations with $10^8 - 10^9$ particles.

References

[1] J. Barnes and P. Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324:446–449, 1986.

[2] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *J. Comp. Phys.*, 73:325–348, 1987.

[3] S. Pfalzner and P. Gibbon. *Many Body Tree Methods in Physics*. Cambridge University Press, New York, 1996.

[4] S. Pfalzner and P. Gibbon. A hierarchical tree code for dense plasma simulation. *Comp. Phys. Commun.*, 79:24–38, 1994.

[5] U. Becciani, V. Antonuccio-Delogu, and M. Gambera. A modifi ed parallel tree code for n-body simulation of the large-scale structure of the universe. *J. Comp. Phys.*, 163:118–132, 2000.

[6] M. S. Warren and J. K. Salmon. A portable parallel particle program. *Comp. Phys. Commun.*, 87(266–290), 1995.

[7] J. Dubinski. A parallel tree code. *New Astronomy*, 1:133–147, 1996.

[8] P. Gibbon, F. N. Beg, R. G. Evans, E. L. Clark, , and M. Zepf. Tree code simulations of proton acceleration from laser-irradiated wire targets. *Phys. Plasmas*, 11:4032–4040, 2004.

[9] S. Pfalzner, S. Umbreit, and Th. Henning. Mass and angular momentum transfer in disc-disc interactions. *Ap. J.*, 629: 526, 2005, submitted.

[10] A. Grama, V. Kumar, and A. Sameh. Scalable parallel formulations of the barnes-hut method for n-body simulations. *Parallel Comp.*, 24:797–822, 1998.

[11] Felix Wolf and Bernd Mohr. Automatic Performance Analysis of Hybrid MPI/OpenMP Applications. *Journal of Systems Architecture, Special Issue 'Evolutions in parallel distributed and network-based processing'*, 49(10–11):421–439, November 2003.

[12] Th. Eickermann, W. Frings, P. Gibbon, L. Kirchakova, D. Mallmann, and A. Visser. Steering UNICORE applications with VISIT. *Phil. Trans. Roy. Soc.*, 363: 1855–1865, 2005.

[13] S. Dominiczak. Development of online visualization for NBODY6++ using the VISIT library. Technical Report FZJ-ZAM-IB-2005-02, Research Centre J¨ulich, 2005. http://www.fz-juelich.de/zam/docs/printable/ib/ib-05/ib-2005-02.pdf

[14] P. Gibbon. PEPC: Pretty Effi cient Parallel Coulomb-solver. ZAM Technical Report FZJ-ZAM-IB-2003-05, Research Centre J¨ulich, 2003. http://www.fz-juelich.de/zam/docs/printable/ib/ib-03/ib-2003-05.pdf

[15] H. Shi and J. Schaeffer. Parallel sorting by regular sampling. *J. Par. Dist. Comp.*, 14:361–372, 1992.

[16] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, Fall 2000.

Experiments with a parallel Monte Carlo simulation of a space cosmic particles detector *

Francisco Almeida[a], Francisco de Sande[a], Carlos Delgado[b] Ramón García–López[b]

[a] Dept. de Estadística, I.O. y Computación Univ. of La Laguna E-38271 La Laguna, Spain

[b] Instituto de Astrofísica de Canarias (IAC) c/Vía Láctea s/n E-38271 La Laguna, Spain

Work in progress in the design of a parallel simulation of a multipurpose cosmic rays detector is presented. The detector is part of an experiment whose main goal is to study the spectrum and composition of charged cosmic rays with unprecedent sensibility.

Parallelism seems to be the most suitable approach to increase the performance of the sequential simulator. Starting from this sequential version of the simulator based on a Monte Carlo method we have developed different parallel codes. Problems related to the random number generation and load balance are the most important issues that we have taken into account.

We present a wide computational experience using the different parallel versions developed and using a PC cluster platform as target architecture.

1. Introduction

Current analysis methods on astroparticle physics [18] require detailed models of the detector response for realistic experimental conditions. Due to the difficulty of the problem this is usually achieved by Monte Carlo simulation, pipelined to the reconstruction software designed for the real detector. This problem has fundamental implications in theoretical physics and astrophysics [9]. The signal search nature of this kind of experiments makes necessary to simulate a number of particles crossing the detector close or larger than the real number of particles expected. This ensures that the statistical uncertainty on any signal is not dominated by the statistical fluctuations of the simulated data.

A huge amount of simulated cosmic rays is needed in advance to understand the detector response. However, the simulation of the particle evolution is very high time consuming in the current sequential code. Furthermore, the nature of the random number generation in Monte Carlo simulations difficult their execution by independent runs using different machines. Parallelism appears as the natural choice to speedup the simulation while keeping control of the random number generation. As stated in [2] the `master-slave` paradigm is a natural fit for parallel Monte Carlo applications. Although Monte Carlo methods are widely regarded as embarrassingly parallel, it is also a well known fact that parallel Monte Carlo applications rely heavily on the availability of statistically independent streams of random numbers to significantly decrease the variance of the calculation. Our parallelization is strongly coerced by the random number generator used in the code. Standard libraries [11] for this purpose can not be used in our case due to the constraint imposed by the potential users of our application who rely on a particular random number generator. Thus, usual parallel techniques dealing with random number generation [19, 3] should be carefully managed.

The remaining of the paper is organized as follows. Section 2 describes the structure, objectives and difficulties found in the design of the sequential simulator. The main challenges involved in the parallelization are presented in section 3. A centralized `master-slave` parallel approach is presented as the first parallelization scheme. In section 4 we present preliminary computational results. Although an important speedup is achieved still some work can be done to reduce the running time of the simulation. We finalize the paper in section 5 with some concluding remarks and future lines of work.

2. The problem

In the experiment under consideration we deal with the first large superconducting cosmic rays detector designed to study an unexplored region of the spectrum. It has very important implications in fundamental physics and astrophysics [9] and is supported by NASA [14] and DOE [6] among others.

We deal with the simulation of a multipurpose particle detector designed to spend a long period of time taking data on space. One of the aims of the experiment is to study the spectrum and composition of charged cosmic rays with unprecedent sensibility. This is achieved by using state of the art detector technology and large amounts of collected data ($\sim 10^{10}$ detected particles are expected). The present simulation code designed by the experiment collaboration is based on the GEANT v3.21 simulation package [17]. In its current state the code execution is fully sequential, so mass production of Monte Carlo data is performed by executing it on a large set of machines. Each simulation is fed with

*This work has been partially supported by the EC (FEDER) and the Spanish MCyT (Plan Nacional de I+D+I, TIC2002-04498-C05-05 and TIC2002-04400-C03-03)

an initial random seed separated enough on the random number sequence of any other seed as to ensure the absence of correlations among different processors.

The simulation code is composed of a main *Triggering* process where the direction and characteristics of the input particles are simulated (Figure 1). This main process is composed of two basic procedures:

- *Propagation (or Tracking):* Simulates the propagation of the particle. This propagation involves the simulation of the interaction with the detector and the recursive propagation and interactions of the resulting particles.
- *Reconstruction:* Simulates the analogical and digital response of the detector to the mentioned interactions. This information is used to reconstruct the properties of the particle generated in the *Triggering* process.

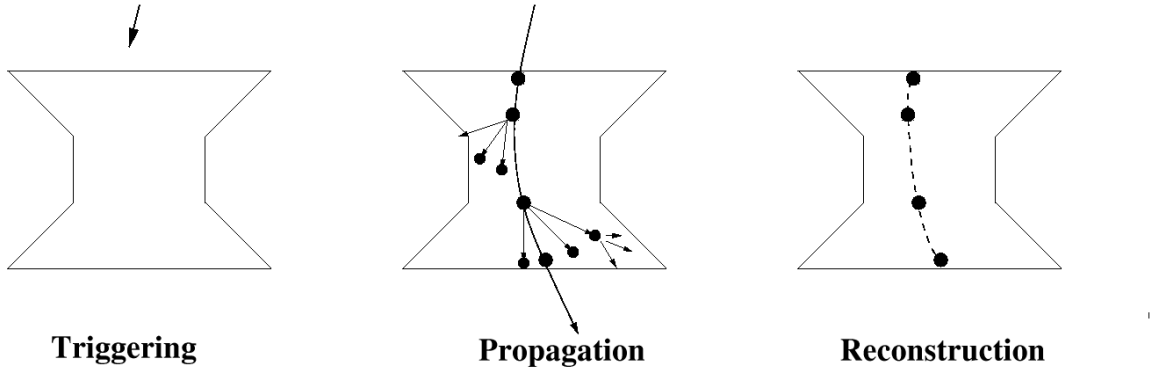


Figure 1. The simulation process

Due to the physics of the interaction of charged particles with matter, the amount of random numbers needed to simulate a single particle, the simulation time and the output size scale roughly as the square of the charge (the number of protons in the nuclei, Z) of the cosmic ray. Taking into account that the simulation time for a charge $Z = 1$ particle ranges from one to several seconds using a single CPU, the current mass production scheme becomes ineffective for simulation of large samples of moderately high charged particles. This ineffectiveness has its roots in the prohibitive long execution time for a small set of incoming cosmic rays using a single machine and in the difficulty for avoiding correlations between different machines due to the amount of random numbers needed.

Parallelization seems to be the adequate solution to the exposed problem for moderate to high Z simulations as long as the core routines, specifically the random number generator and physics, are kept unchanged, as they have been tested for years independently in many particle physics experiments at CERN [1, 5, 8, 15]. To deal with the execution time problem, the solution has to be such that the work is spread over a number of processors keeping the consistency of the random numbers sequence. Moreover it has to manage a large quantity of processed data per particle, which amounts up to several tens of Kb for $Z = 1$ cosmic rays. Due to the difficulties involved in the development of such code, there is no previous parallel version of the simulation available at this moment.

Finally a last requirement has to be met: the parallel solution has to be available as soon as possible since the Monte Carlo data production should be large enough before the beginning of the physical data acquisition. A large amount of high Z simulated cosmic rays is needed in advance to understand the detector response.

3. Parallelization

From the high performance computing point of view, the parallelization of the code is a challenge because it involves many different issues that make it difficult:

- It is a *real life* code and therefore the code size is not easy to manage (about 200000 lines).
- It is a collaborative work where many programmers have been applied, and therefore the structure of the application is not clear even for some one not involved in the development, even if he/she is an expert programmer.
- The code is written using two different programming languages: it mixes C++ and Fortran77.

- The physical simulation requires very specialized knowledge in the field, not easy to achieve for a computer scientist: an interdisciplinary effort is mandatory.
- The sequential version of the code was provided by a team belonging to the experiment, and the parallelization has to be done under the condition that none of the low level numeric routines can be modified, as they have been certified by the project only in its current form. In particular the random number generator has to be treated as a black box, which has important consequences in the parallelization, as we will show later.
- The program uses several external libraries (more than 10 in its simplest mode), including threaded code. The incorporation of the parallel context has to be carefully analyzed to avoid non desired interactions.
- The code is highly sensitive to numerical arithmetic precision, so that it strongly relies on the compilers used.
- The very large amount of data that have to be managed by the parallel code is also a challenge. One terabyte of data can be produced by the simulator for an average input.

These difficulties force us to avoid the parallel design starting from scratch. One of our constraints is to introduce the minimum amount of changes in the original code. Once the most time consuming code sections have been identified, our goal is to parallelize the application intervening only on them.

In the last years OpenMP [16] and MPI [13] have been universally accepted as the standard tools to develop parallel applications. Some reasons brought us to discard OpenMP and the shared memory model as the initial option for our parallelization:

- To reach acceptable performance results we foresee that we will need a huge number of processors. Nowadays this is a limitation for shared memory architectures.
- To precisely identify the accessibility of the variables in the code is a difficulty when dealing with such large amount of data. An extra complication comes from the high number of Fortran `common` blocks in the program.
- The nature of the code makes it sensitive to load imbalance. We wonder whether to use the OpenMP dynamic schedule clause is mature enough to deal with this problem efficiently [4].

In the parallelization of the code, we are following a two level incremental approach. In the first level we will broach the different sections of the code independently, while in the second level we will focus our attention in different target platforms. Currently we have parallelized the *Triggering* section and several versions for PC clusters are available. *Propagation* and *Reconstruction* sections of the code are also candidates to be parallelized. The parallel platforms that we are considering for future development range from PC clusters and proprietary parallel platforms to grid resources, combining both of them.

Figure 2 illustrates the structure of the sequential code and the parallelization of its *Triggering* section. We have followed a centralized `master-slave` approach, where the `master` tracks the subproblem generation and takes charge of the load balance. The `collector` process manages the huge amount of output data generated by the `slaves`. In the scheme depicted for the parallel *Triggering* process in Figure 2, the `slave` processes track the interactions produced by one particle when it goes through the detector producing new particles. This process intensively demands random numbers to the `rnd generator` process and the result of these interactions are delivered to the `collector` process.

Special consideration has to be deserved to the random number generation for the Monte Carlo simulation. There are two well known families of techniques to construct parallel pseudorandom number generators from a sequential one. The *parameterization* methods (see [12], for example) consist in parameterize different branches of pseudorandom number sequences from the same generator, such that each node uses a different one. The *cycle division* methods (see [10], for example) take a single pseudorandom number sequence, which is partitioned in a unique sequence for each node. As the sequential random number generator has to be treated as a black box, in the sense that the routine can be called but not modified, the *parameterization* is somehow unrealizable, so we have to stick to *cycle division* techniques. On the other hand the huge amount of random numbers needed ($\sim 1,6 \times 10^6$ numbers per particle) suggest to follow a strategy that overlaps computation and communications. This should allow that the `slaves` get the random numbers as soon as they are required. In order to avoid anomalies in the convergence of the simulation, our first approach was to use a centralized random number generator, however we also tried a distributed strategy.

We describe now the different parallel versions currently developed: In the first approach, the master process deals with two functionalities: random number generation and distribution of tasks. We will refer this variant as C (Centralized). In the next version, the task distribution function and the random number generation run in different processors. Two different approaches have been developed for this variant: in the first one, the communications are synchronous

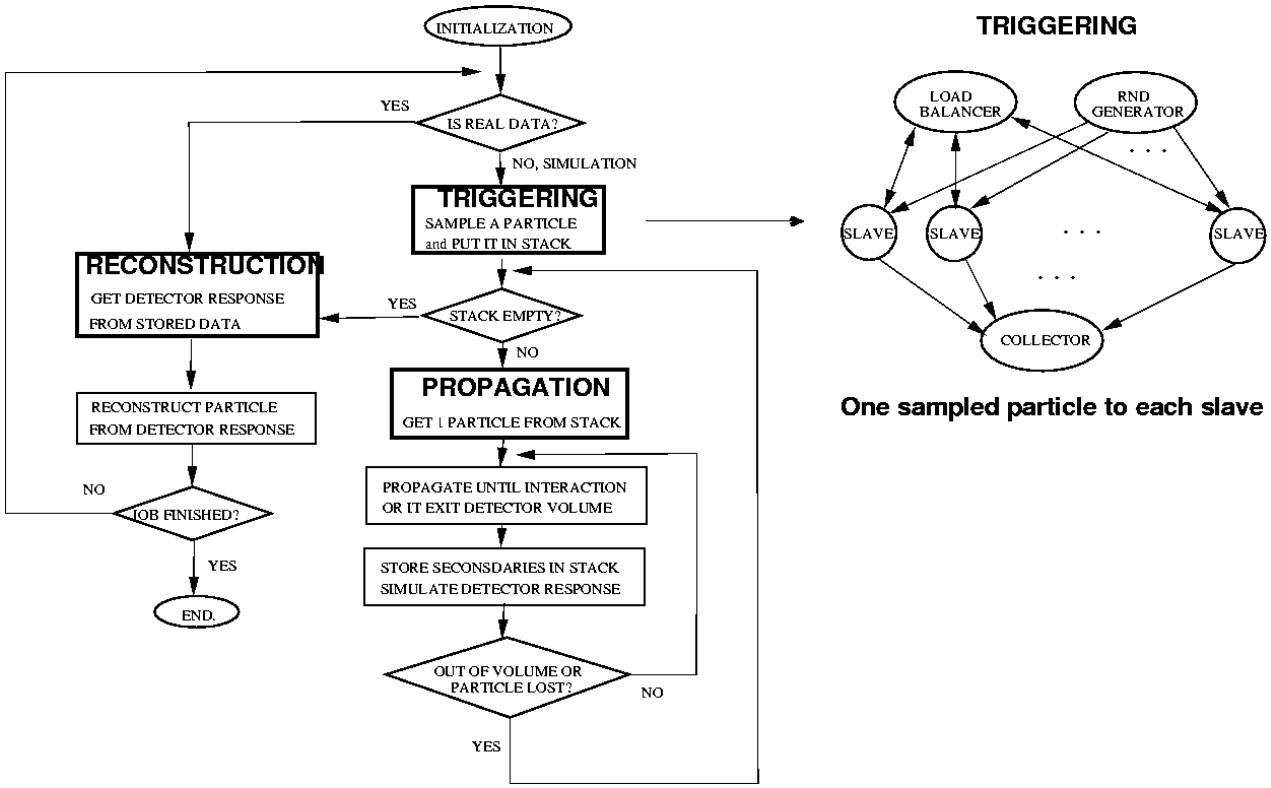


Figure 2. Structure of the parallel `master-slave`.

(each slave requests a package of random numbers to the server when needed). In the asynchronous approach the packages are sent to the slaves in advance, without any request. These versions are referred as CCS (both functions are centralized (C) and use synchronous mode) and CCA (both functions are centralized and use asynchronous mode). Finally, in the CD (Centralized task generation and distributed random number generation), each slave follows a leapfrog [19] strategy for random number generation.

4. Computational Results

In this section we report the performance achieved for the different parallel versions of the simulation on a PC cluster platform using the `mpich` [7] implementation of MPI. The computational experiments have been carried out in a parallel machine with 16 nodes. Each node is a 2 GB memory shared memory bi-processor composed of Intel Xeon 2.80GHz processors interconnected with a 100 Mbit/sec. fast Ethernet switch and running Linux.

Table 1
Running Times (secs.) and Speedups for the centralized (C) parallel version and for the tracking procedure.

#procs.	Running Time	Tracking Time	Total Speedup	Tracking Speedup
1	35623	12126		
2	25799	13011	1.38	0.93
6	9284	4154	3.83	2.91
11	4663	2166	7.63	5.59
16	3191	1117	11.16	10.85
21	2584	1022	13.78	11.86
26	2943	880	12.10	13.77
31	2604	585	13.67	20.69

We have simulated 10000 He4 nuclei (alpha radiation) impinging onto the detector with an uniform spatial distribu-

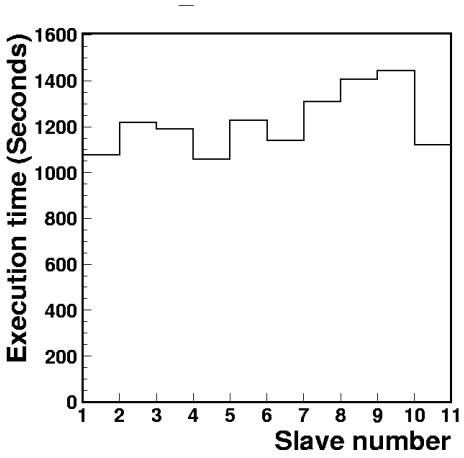


Figure 3. Load balance in the *Tracking* procedure using 12 processors.

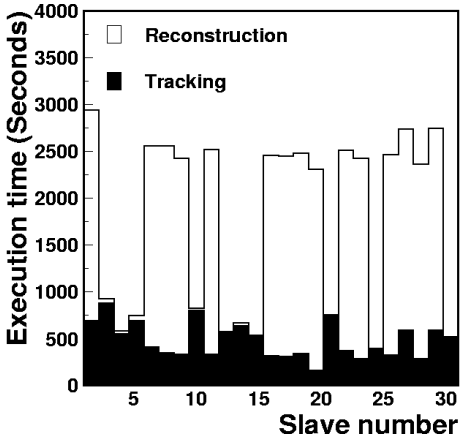


Figure 4. The load balance for the *Tracking* and *Reconstruction* using 32 processors.

tion and with an isotropic distribution on directions. The simulated particles have an initial momentum ranging from 400GeV/c to 3700GeV/c with an exponential decaying distribution. The simulated flux is equivalent to several minutes of real data taking of the detector at a height of 350 Km over the sea level.

The purpose of these experiments is not only to focus on the speedup achieved but to understand the sources of overhead in the parallel versions. The analysis studies the load imbalance introduced for each of the procedures involved in the *Triggering* process. As usual, we measure the sequential running time for the *Triggering* process and compare it with the parallel running times. To understand the behaviour of the parallel program we also isolated the running times for the *Tracking* procedure and measure the speedups obtained.

Table 1 and Figure 5 show these running times and speedups. We observe how the parallel approach introduces an important reduction on the running time of the *Tracking* procedure. The speedup increases with the number of processors. An speedup of `20.69` is achieved when using `31` slaves. However, a maximum speedup of `13.78` is obtained for the global simulation process when using `21` slaves. No increment is observed after this point when the number of processors increase.

Figure 3 shows the load imbalance introduced in the parallelization of the *Tracking* procedure when using `11` slaves. The load imbalance introduced is no higher than a 20%. We consider it as acceptable. However, the load imbalance introduced in the *Reconstruction* phase is prohibitive (Figure 4). This is the main reason for the low performance obtained with a large number of processors. For some $anomalous$ events, the *Reconstruction* process degenerates and consumes all the CPU time of the assigned processor. This is a well known phenomenon of physical nature that also appears in the sequential simulation: A huge amount of secondary particles are generated in the *Tracking* stage. This fact introduces a large overhead on the combinatorial reconstruction that grows exponentially. The number of $anomalous$

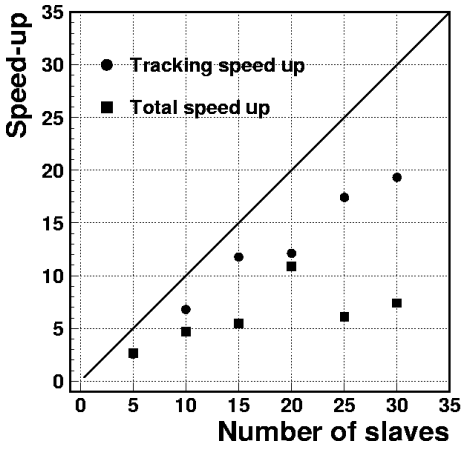


Figure 5. Speedup curves for the global simulation process and for the *Tracking* procedure. Centralized (C) version.

events in the parallel simulation is much higher than in the sequential one.

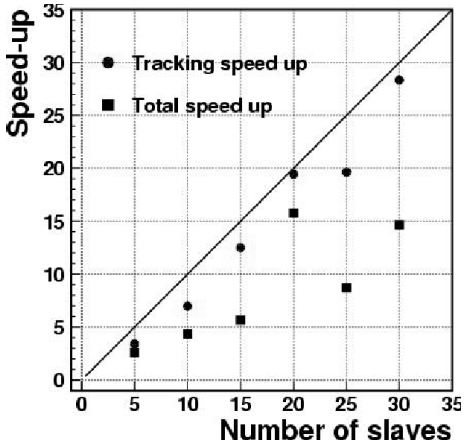


Figure 6. Speedup curves for the global simulation process and for the *Tracking* procedure. CCS version.

In the CCS version the random number generation and task generation are decoupled in different processors. The results of this version are shown in Figure 6 and we can appreciate an improvement in the speedup for the *Tracking* procedure that is almost linear. As a consequence the total speedup of the program is also improved.

The CCA version, with asynchronous communications whose results are depicted in Figure 7 did not deliver the expected performance. Although its results are better than those of the C version, it performs worse than the CCS version.

Finally, Figure 8 shows the results obtained for the CD version, using distributed random number generation. We observe that this version is the one offering poorer performance. Additional experimentation showed us that the time consumed by the random number generation was very high, being this the main reason for the low performance observed.

5. Conclusions

We conclude that the scientific aim of applying high performance computing to computationally-intensive codes in particle physics is being successfully achieved. The importance of the results we have obtained come both from the scientific relevance of this code and also from stating that parallel computing techniques are the key to broach large size real problems in the mentioned scientific field. A wide computational experience has been developed and an important reduction of the sequential running time is obtained with the number of processors considered.

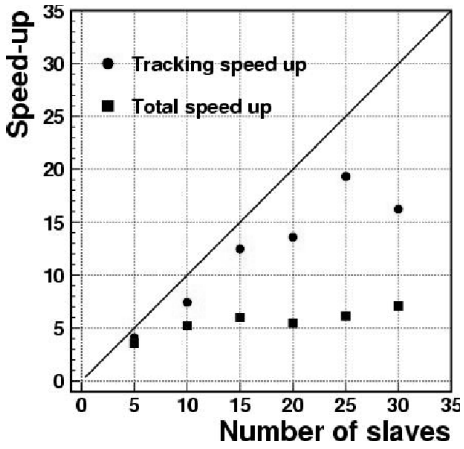


Figure 7. Speedup curves for the global simulation process and for the *Tracking* procedure. CCA version.

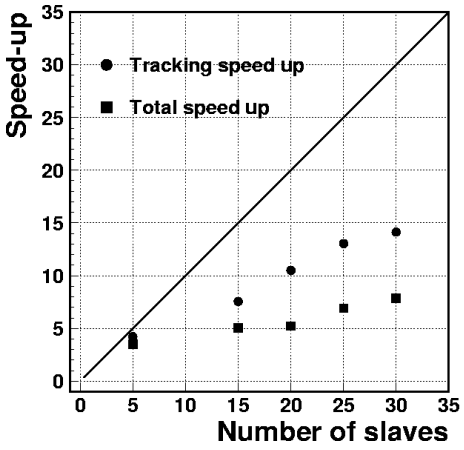


Figure 8. Speedup curves for the global simulation process and for the *Tracking* procedure. CD version.

Although the best results for our preliminary target architecture are delivered by the CCS parallel version of the code, we believe that future experiments in different platforms could reveal different behaviours.

The sources of overhead in some of the parallel codes will be deeper analyzed in the near future and new levels of parallelization will be introduced. With these objectives in mind, we look out for an improved parallel version scaling to a much larger number of processors.

References

[1] ALEPH Collab. *Nucl. Inst. and Meth. A*, (294):121, 1990.

[2] Jim Basney, Rajesh Raman, and Miron Livny. High Throughput Monte Carlo. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, TX, March 1999.

[3] B. C. Bromley. Quasirandom number generators for parallel Monte Carlo algorithms. *Journal of Parallel and Distributed Computing*, 38(1):101–104, 1996.

[4] J. M. Bull. Measuring synchronisation and scheduling overheads in OpenMP. In *Proc. First European Workshop on OpenMP (EWOMP 1999)*, pages 99–105, Lund, Sweden, Sep 1999.

[5] DELPHI Collab. *Nucl. Inst. and Meth. A*, (303):233, 1991.

[6] U.S. Department of Energy. `http://www.energy.gov/engine/content.do`.

[7] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[8] L3 Collab. *Nucl. Inst. and Meth. A*, (289):35, 1990.

[9] M. S. Longair. *High Energy Astrophysics*. Cambridge University Press, Cambridge, England, 1992.

[10] J. Makino. Lagged-fibonacci random number generator on parallel computers. *Parallel Computing*, 20:1357–1367, 1994.
[11] Mascagni and Srinivasan. SPRNG: A scalable library for pseudorandom number generation. *ACMTMS: ACM Transactions on Mathematical Software*, 26, 2000.
[12] M. Mascagni and M.L. Robinson S.A. Cuccaro, D.V. Pryor. A fast, high quality, and reproducible lagged-fibonacci pseudorandom number generator. *Journal of Computational Physics*, 15:211–219, 1995.
[13] MPI Forum. The MPI standard. `http://www.mpi-forum.org/`.
[14] National Aeronautics and Space Administration. `http://www.nasa.gov/`.
[15] OPAL Collab. *Nucl. Inst. and Meth. A*, (305):275, 1991.
[16] OpenMP Architecture Review Board. *OpenMP Fortran Application Program Interface*. OpenMP Forum, Nov 2000. `http://www.openmp.org/specs/mp-documents/fspec20.pdf`.
[17] R. Brun et. al. *GEANT Users Guide*. CERN, 1994.
[18] S. Eidelman et al. Review of Particle Physics. *Physics Letters B*, 592:1+, 2004.
[19] Ashok Srinivasan, Michael Mascagni, and David Ceperley. Testing parallel random number generators. *Parallel Comput.*, 29(1):69–94, 2003.

Parallel Simulation of Tsunamis Using a Hybrid Software Approach

X. Cai[ab], G. K. Pedersen[c], H. P. Langtangen[ab], S. Glimsdal[ab]

[a]Simula Research Laboratory, P.O. Box 134, N-1325 Lysaker, Norway

[b]Department of Informatics, University of Oslo, P.O. Box 1080, Blindern, N-0316 Oslo, Norway

[c]Department of Mathematics, Mechanics Division, University of Oslo, P.O. Box 1053, Blindern, N-0316 Oslo, Norway

Simulation of tsunamis in large ocean domains, such as the Indian Ocean, presents a huge computational and software challenge, which the traditional parallel software alone can not meet. To effectively use the computational resources, we should allow different mathematical models, different numerical methods, and different mesh types/resolutions in different areas of the vast ocean domain. This ensures that complex mathematical models and high mesh resolution are only used where necessary. Such a parallel hybrid tsunami simulator calls for a very flexible software approach that can mix different models, methods and meshes, maybe even incorporate "alien software". This paper thus explains how this can be achieved by combining overlapping domain decomposition and object-oriented programming. Some preliminary simulation results of the Indian Ocean Tsunami are also provided to demonstrate the applicability of the proposed software approach.

1. Introduction

The tragic event of the Indian Ocean Tsunami on December 26, 2004 has once again grimly reminded us about the importance of tsunami modeling and simulation. This challenging research field spans a wide spectrum and involves many interplayed topics, such as earthquake modeling, ocean wave propagation and coastal modeling. The main focus of the present paper is to investigate how to effectively simulate the propagation of tsunami over a vast ocean. Although the topic of water wave propagation has been subject to extensive research, the particular case of simulating the Indian Ocean Tsunami presents a new challenge because of the huge size of the computations. Moreover, many complex features need to be considered in ocean wave propagation, such as the locally rapidly changing bathymetry, dispersion, nonlinear effects, and complicatedly shaped coastlines.

To obtain a balance between numerical accuracy and computational efficiency, we should only apply advanced numerical techniques and high mesh resolutions to small areas where necessary, while resorting to simpler numerical techniques and coarser meshes in the remaining areas. In terms of mathematical models, this means a choice between the most widely used linear wave equations and the more complex Boussinesq wave equations, which involve both weak dispersion and nonlinear effects. Perhaps also the Navier-Stokes equations may sometimes be used in certain small areas. In terms of numerics, there is a choice between finite differences, finite volumes, and finite elements. The finite difference method (FDM) results in the fastest computations, at least on a uniform mesh of rectangular shape. The finite volume method (FVM) or the finite element method (FEM), on the other hand, run slower but are better adapted to unstructured meshes and resolution variations needed for treating, e.g., areas of rapidly changing ocean bottom topography.

Considering the fact that there already exist many software packages for wave simulation, and that parallel computing must be used for detailed ocean modeling, an ideal tsunami simulator should be built by a hybrid software approach. More specifically, the entire ocean domain is divided into many

subdomains, and independent solvers are assigned to the subdomains. Each subdomain solver has its own mathematical model, together with the matching numerical method and subdomain mesh. Therefore, different subdomains may use different mathematical models, different numerical methods, different meshes and even different software! Of course, to make the above hybrid divide-and-conquer simulator to work, we need a numerical framework overseeing all the subdomains. This can be achieved by the overlapping domain decomposition (DD) strategy [5,15], which was originally designed for using a same mathematical model everywhere, but can be extended to act as an "umbrella" for the subdomains in the hybrid tsunami simulator. The communication between the subdomains is controlled by the DD framework and implemented as exchanging messages using MPI [7,10]. In respect of programming, the framework of DD can be implemented in an object-oriented style [1], making it easier to encompass different types of subdomain solvers and to adopt "alien software" when necessary.

The remaining text of the paper is organized as follows. Section 2 explains two mathematical models commonly used for simulating wave propagation and the additive Schwarz scheme needed as the numerical foundation of the parallel hybrid tsunami simulator. Section 3 then concentrates on the implementation aspect, based on object-oriented programming. Thereafter, Section 4 presents some preliminary parallel simulation results of the Indian Ocean Tsunami. Finally, some concluding remarks and comments about future work are given in Section 5.

2. Mathematics and Numerics

2.1. Boussinesq Water Wave Equations

The nonlinear Boussinesq water wave equations can be used to simulate ocean waves; see e.g. [14, 11,2,16]. In comparison with the standard linear wave equations, the Boussinesq equations can model weakly dispersive and nonlinear waves. There exist several variants of the Boussinesq equations, among which we will consider the following two coupled partial differential equations:

$$\frac{\partial \eta}{\partial t} + \nabla \cdot (H + \alpha\eta)\nabla\phi + \epsilon H \left(\frac{1}{6}\frac{\partial \eta}{\partial t} - \frac{1}{3}\nabla H \cdot \nabla\phi \right) \nabla H = 0, \tag{1}$$

$$\frac{\partial \phi}{\partial t} + \frac{\alpha}{2}\nabla\phi \cdot \nabla\phi + \eta - \frac{\epsilon}{2} H \nabla \cdot \left(H \nabla \frac{\partial \phi}{\partial t} \right) + \frac{\epsilon}{6} H^2 \nabla^2 \frac{\partial \phi}{\partial t} = 0, \tag{2}$$

where the primary unknowns are the surface elevation $\eta(x, y, t)$ and the depth averaged velocity potential $\phi(x, y, t)$. In the above equations $H(x, y)$ denotes the water depth, and the constants ϵ and α are used to control the magnitude of dispersion and nonlinearity, respectively. With $\epsilon = \alpha = 0$ we recover the standard linear wave equations from (1)-(2).

A standard numerical strategy for solving (1)-(2) first carries out the temporal discretization, with centered differences on a staggered grid in time [13]:

$$\frac{\eta^{\ell} - \eta^{\ell-1}}{\Delta t} + \nabla \cdot \left(\left(H + \alpha \frac{\eta^{\ell-1} + \eta^{\ell}}{2} \right) \nabla \phi^{\ell-\frac{1}{2}} + \epsilon H \left(\frac{1}{6}\frac{\eta^{\ell} - \eta^{\ell-1}}{\Delta t} - \frac{1}{3}\nabla H \cdot \nabla \phi^{\ell-\frac{1}{2}} \right) \nabla H \right) = 0,$$

$$\frac{\phi^{\ell+\frac{1}{2}} - \phi^{\ell-\frac{1}{2}}}{\Delta t} + \frac{\alpha}{2}\nabla\phi^{\ell-\frac{1}{2}} \cdot \nabla\phi^{\ell+\frac{1}{2}} - \frac{\epsilon H}{2}\nabla \cdot \left(H \frac{\nabla\phi^{\ell+\frac{1}{2}} - \nabla\phi^{\ell-\frac{1}{2}}}{\Delta t} \right) + \frac{\epsilon H^2}{6}\frac{\nabla^2\phi^{\ell+\frac{1}{2}} - \nabla^2\phi^{\ell-\frac{1}{2}}}{\Delta t} = -\eta^{\ell}.$$

Here, index ℓ denotes the discrete time levels and Δt is the time step size. Note that η is sought at integer time levels (ℓ) and ϕ is sought at half-integer time levels ($\ell + \frac{1}{2}$). The remaining part of the numerical scheme is to carry out the spatial discretization of the above two semi-discretized equations at each time step, using FDM or FEM, and solve for η^{ℓ} and $\phi^{\ell+\frac{1}{2}}$. The readers are referred to [9,4] for more details.

2.2. The Parallel Multi-Subdomain Strategy

As we can see in the preceding text, the computational task at time step ℓ is to find η^{ℓ} and $\phi^{\ell+\frac{1}{2}}$ based on the solutions from the previous time step: $\eta^{\ell-1}$ and $\phi^{\ell-\frac{1}{2}}$. To incorporate parallelism, we use the additive Schwarz scheme [5,15], which is an overlapping DD method. The entire ocean domain Ω is first decomposed into a set of *overlapping* subdomains Ω_s, $1 \leq s \leq P$. Overlapping zones are present between neighboring subdomains. In the context of solving discretized Boussinesq equations at time step ℓ, the additive Schwarz scheme is transformed into the following parallel numerical strategy:

- Set $\eta_s^{\ell,0} = \eta^{\ell-1}|_{\Omega_s}$, where $\eta^{\ell-1}|_{\Omega_s}$ denotes the restriction of the global solution $\eta^{\ell-1}$ (from time step $\ell-1$) onto subdomain s.
- Carry out the following Schwarz iterations for $k = 1, 2, 3, \ldots$ until convergence of η among the subdomains:
  1. On each subdomain find an improved local solution $\eta_s^{\ell,k}$ based on $\eta_s^{\ell,k-1}$ and $\phi^{\ell-\frac{1}{2}}|_{\Omega_s}$.
  2. Compose a temporary global solution $\eta^{\ell,k}$ by "sewing together" the latest subdomain solutions $\left\{\eta_s^{\ell,k}\right\}$. In each overlapping zone between two or more neighboring subdomains, averaging between different subdomain solutions is enforced. In case neighboring subdomains have different mesh resolutions in an overlapping zone, interpolation between the subdomain meshes is used in the averaging.
- Set $\phi_s^{\ell+\frac{1}{2},0} = \phi^{\ell-\frac{1}{2}}|_{\Omega_s}$ and carry out the Schwarz iterations with respect to ϕ as above.

It should be noted that the above numerical strategy extends the multi-subdomain strategy proposed in [4]. Here, the adopted mathematical model and/or numerical method may differ from subdomain to subdomain, so are the type and resolution of the subdomain meshes. During each Schwarz iteration, the process of solving $\eta_s^{\ell,k}$ and $\phi_s^{\ell+\frac{1}{2},k}$ on subdomain s is decided by the subdomain solver independently. For example, some subdomains may adopt the linear wave equations, i.e., $\epsilon = \alpha = 0$ in (1)-(2), and thus find $\eta_s^{\ell,k}$ and $\phi_s^{\ell+\frac{1}{2},k}$ by an explicit updating scheme, whereas other subdomains may consider dispersion and/or nonlinearity in the Boussinesq equations and thus need an implicit solution scheme. The above parallel multi-subdomain strategy also differs from the classical additive Schwarz method [5,15] in that there does not always exist a global linear system coupling all $\eta_s^{\ell,k}$ or $\phi_s^{\ell+\frac{1}{2},k}$.

Although the subdomain solvers are mostly independent of each other, exchange of subdomain solutions within the overlapping zones is necessary for obtaining global convergence. Thus a global administrator is needed to synchronize the pace of the subdomain solvers during the task of "sewing together" subdomain solutions at the end of each Schwarz iteration. Since different subdomains normally reside on different processors, the global administrator initiates the inter-processor communication in form of message passing. No subdomain is allowed to proceed to the next Schwarz iteration before all its neighbors have received needed information.

Checking the convergence of the Schwarz iterations is another major task of the global administrator. More specifically, after the message passing phase is finished at the end of each Schwarz iteration, each subdomain checks the difference between $\eta^{\ell,k}|_{\Omega_s}$ and $\eta^{\ell,k-1}|_{\Omega_s}$ (recall that $\eta^{\ell,k}$ is the result of "sewing together" $\eta_s^{\ell,k}$ from all the subdomains). If $\left\|\eta^{\ell,k}|_{\Omega_s} - \eta^{\ell,k-1}|_{\Omega_s}\right\|$ is small enough, subdomain s sends a flag indicating local convergence to the global administrator. Otherwise a flag

of no local convergence is sent. It is only when *all* the subdomains have reported local convergence that the global administrator deems that global convergence is reached and stops the Schwarz iterations.

3. A Parallel Hybrid Tsunami Simulator

3.1. Making Use of a Generic DD Framework

As has been explained in Section 2.2, the multi-subdomain parallelization strategy uses additive Schwarz iterations as the numerical foundation. It should be noted that such a parallelization strategy is not only applicable to tsunami simulation, but is generic for building many other parallel partial differential equation solvers. Therefore, object-oriented programming has been adopted to implement a parallel DD framework, reusable for many occasions. For details we refer the readers to [3], and it suffices for the present paper to say that the global administrator (see Section 2.2) can be implemented as class `Administrator` and a generic subdomain solver as class `SubdomainSolver`. The essence is that common tasks, such as domain partitioning, inter-subdomain communication and control of Schwarz iterations, are implemented as member functions in `Administrator`, whereas `SubdomainSolver` defines a set of virtual member functions constituting a generic interface of any particular subdomain solver. MPI is used inside `Administrator` for inter-subdomain communication, such that the generic DD framework is portable on all parallel computers.

3.2. Coupling FDM and FEM in a Hybrid Simulator

We have, as the starting point for our parallel hybrid tsunami simulator, two different pieces of serial software: a flexible Boussinesq solver written in C++ and a legacy Fortran 77 code. The C++ Boussinesq solver is implemented as class `Boussinesq` in the Diffpack programming environment [6,12]. This C++ solver uses finite elements and handles both unstructured and uniform meshes. When dispersion and/or nonlinearity are considered in (1)-(2), the two resulting linear systems at each time step can be solved by a variety of linear solvers provided by Diffpack. The legacy Fortran 77 code is a set of subroutines which are much less flexible in that only uniform meshes are allowed, the equations are discretized by FDM, and the linear solver is fixed as the alternating line-version of the SSOR method. However, the main advantage of the Fortran 77 code is its computational efficiency. The code is also reliable and well tested over two decades.

To incorporate both serial Boussinesq solvers as subdomain solvers into a parallel hybrid tsunami simulator, we have created two light-weight new classes:

`SubdomainBQFEMSolver` and `SubdomainBQFDMSolver`

Here, class `SubdomainBQFEMSolver` is implemented as subclass of both `SubdomainSolver` and `Boussinesq`, so that it inherits the computational functionality from `Boussinesq` while becoming recognizable by the generic `Administrator` as a subdomain solver in the generic DD framework. Similarly, class `SubdomainBQFDMSolver` is derived from `SubdomainSolver` and acts as a wrapper of the Fortran 77 subroutines. Finally, another new class `HybridBQSolver` is derived as subclass of `Administrator`, so that some tsunami specific functionality can be added on top of generic DD functionality.

4. Preliminary Results of Indian Ocean Tsunami

In this section we present some preliminary simulation results of the Indian Ocean Tsunami on December 26, 2004. It should be emphasized that the following results are only meant to demonstrate

the applicability of the hybrid software approach to tsunami simulation. Only relatively low mesh resolution has been used so far. Ongoing research is currently testing the suitable mathematical model and mesh resolution in different regions of the Indian Ocean, in preparation of doing really large-scale parallel hybrid tsunami simulations.

4.1. The Ocean Domain

Our computational domain Ω covers the entire Indian Ocean from the coastlines of Sumatra in the west to the African coastlines in the east. The size of Ω is 8096.08km$\times 4777.65$km. Figure 1 shows a plot of the ocean bottom topography $H(x, y)$, built from publicly available bathymetry data.

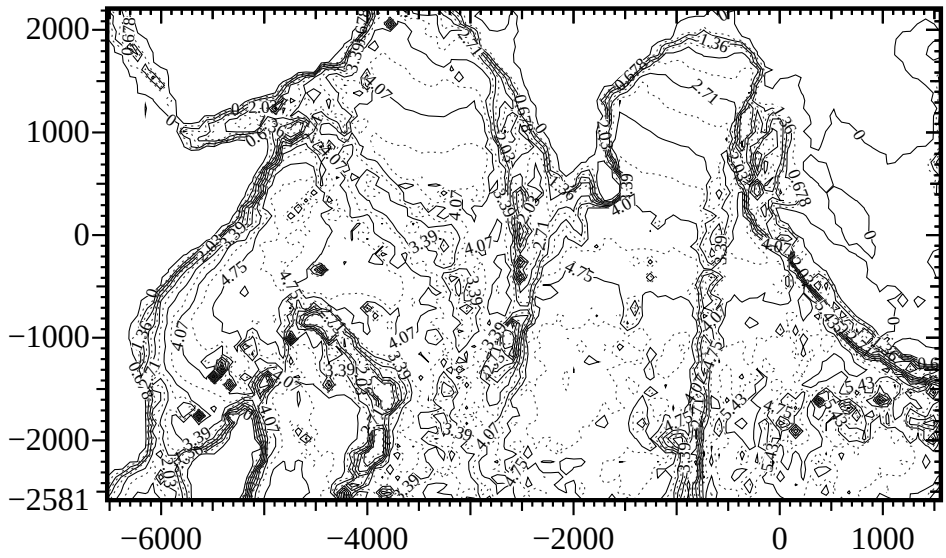


Figure 1. Contour curves representing the water depth (in kilometers) of the entire solution domain. The position $(0, 0)$ represents the epicenter of the submarine earthquake that generated the tsunami.

4.2. Simulation on a Global Uniform Mesh

As a simple test case we use a coarse uniform 1093×645 mesh, where each cell is approximately 7.4km$\times$7.4km. The entire solution domain is divided into $4 \times 4 = 16$ subdomains, which all use the Fortran 77 FDM code as the subdomain solver. Only dispersion is considered, i.e., $\epsilon = 1$ and $\alpha = 0$ in (1)-(2). Figure 2 shows a particular initial condition (see [8]) and two snapshots of the simulated η solution in a zoomed-in region around the epicenter.

4.3. Adaptive Mesh Refinement in the Malacca Strait

Close to the epicenter of the earthquake, such as in the Malacca Strait, the water is extremely shallow, so higher mesh resolution is obviously needed. To test this possibility, we carry out local adaptive mesh refinement in a focused rectangular region around the epicenter. The left plot in Figure 3 shows a zoomed-in plot of the resulting unstructured finite element mesh. The focused region after the adaptive mesh refinement is further partitioned into 17 smaller subdomains that use `SubdomainBQFEMSolver`. The remaining 15 subdomains use `SubdomainBQFDMSolver`, as in Section 4.2. Inside the overlapping zones, the mesh points of the FEM subdomains may not match those of the neighboring FDM subdomains, thus interpolation is required when `HybridBQSolver` "sews together" the subdomain solutions at the end of each Schwarz iteration, see Section 2.2. A

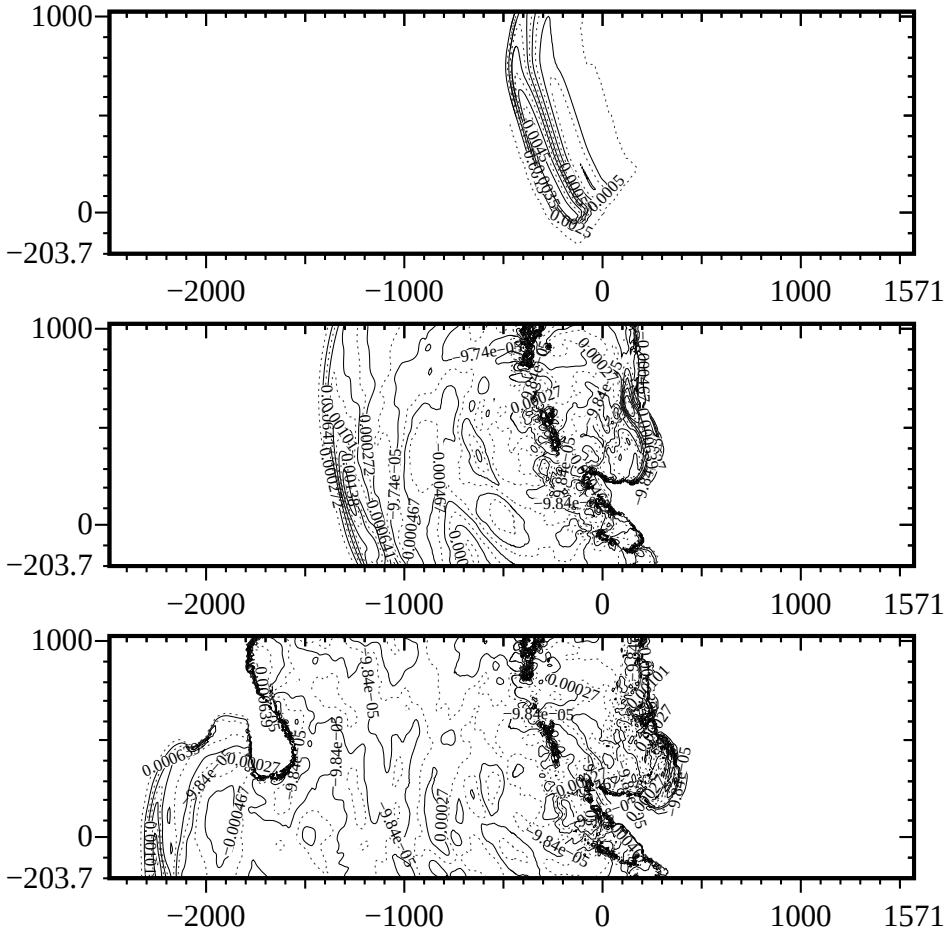


Figure 2. Contour curves of the η solution that are zoomed into a region around the epicenter. Top plot: initial condition. Middle plot: simulated η after approximately 1.4 hours. Bottom plot: simulated η after approximately 2.8 hours.

new simulation is thus run using this setup of hybrid subdomain solvers. Figure 4 shows a snapshot of η approximately 2.8 hours after the earthquake. In comparison with the bottom plot in Figure 2, we can see that adaptively refined finite element meshes produce more local details of η around the epicenter.

5. Concluding Remarks

The simulations reported in Section 4 are rather a proof of concept for the hybrid tsunami simulator. They demonstrate that the resulting parallel simulator is capable of adopting different numerical methods and subdomain meshes in different areas of the ocean domain. We emphasize once again that overlapping domain decomposition and object-oriented programming are the two main ingredients in the hybrid simulator. Meaningful simulations will be carried out in the future using a much finer mesh resolution than that used in Section 4. Nevertheless, for most areas where uniform subdomain meshes are appropriate, FDM should be used for the computational efficiency. For other areas, such as in shallow regions and near the coastlines, locally unstructured finite element fine meshes must be used, thus requiring FEM in the subdomain solvers.

Since a FEM solver is typically an order of magnitude slower than a FDM solver, more (and

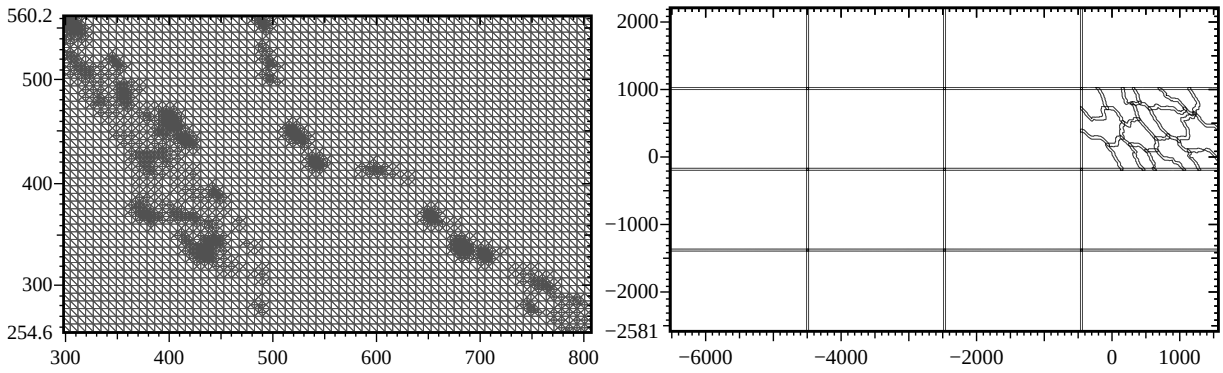


Figure 3. Adaptively refined finite element mesh in a zoomed-in area in the Malacca Strait, and an unstructured repartitioning of the region where adaptive mesh refinement has been carried out.

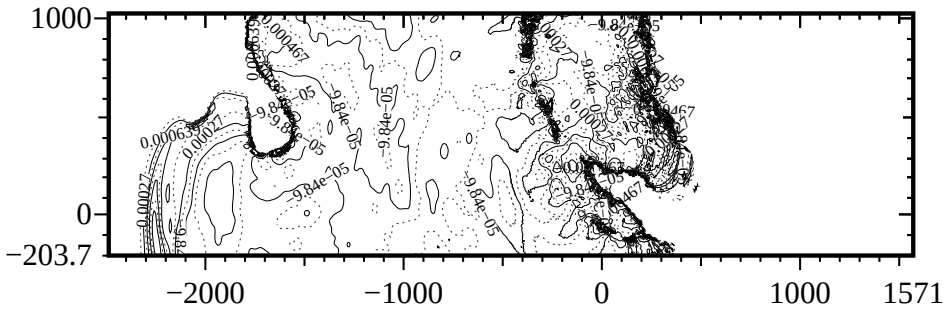


Figure 4. Contour curves of the simulated η solution, zoomed into an area around the epicenter; approximately 2.8 hours after the earthquake.

smaller) subdomains should be used in areas with locally refined meshes, see the right plot in Figure 3. This will help to maintain a reasonable level of load balance between the subdomains. More precisely, a few test time steps can be run on a relatively small number of processors and the CPU time is measured on each subdomain. The CPU time ratio between a FEM subdomain and a standard FDM subdomain gives the number of smaller subdomains into which the FEM subdomain should be further decomposed. When all the FEM subdomains are further decomposed, production simulations can be run on the newly extended set of subdomains, which may differ greatly in terms of area size and number of mesh points but are relatively balanced with respect to the computational speed.

References

[1] J. J. Barton and L. R. Nackman. *Scientific and Engineering C++ – An Introduction with Advanced Techniques and Examples.* Addison-Wesley, 1994.

[2] Marine Accident Investigation Branch. Report on the investigation of the man overboard fatality from the angling boat Purdy at Shipwash Bank off Harwich on 17 July 1999. Technical Report 17/2000, Marine Accident Investigation Branch, Carlton House, Carlton Place, Southampton, SO15 2DZ, 2000.

[3] X. Cai. Overlapping domain decomposition methods. In H. P. Langtangen and A. Tveito, editors, *Advanced Topics in Computational Partial Differential Equations – Numerical Methods and Diffpack Programming*, pages 57–95. Springer, 2003.

[4] X. Cai, G. K. Pedersen, and H. P. Langtangen. A parallel multi-subdomain strategy for solving Boussi-

nesq water wave equations. *Advances in Water Resources*, 28:215–233, 2005.

[5] T. F. Chan and T. P. Mathew. Domain decomposition algorithms. In *Acta Numerica 1994*, pages 61–143. Cambridge University Press, 1994.

[6] Diffpack Home Page. http://www.diffpack.com.

[7] Message Passing Interface Forum. MPI: A message-passing interface standard. *Internat. J. Supercomputer Appl.*, 8:159–416, 1994.

[8] S. Glimsdal, G. Pedersen, K. Atakan, C. B. Harbitz, H. P. Langtangen, and F. Løvholt. Propagation of the Dec. 26, 2004 Indian Ocean Tsunami: Effects of dispersion and source characteristics. *Int. J. Fluid Mech. Research*, 2005. To appear.

[9] S. Glimsdal, G. K. Pedersen, and H. P. Langtangen. An investigation of overlapping domain decomposition methods for one-dimensional dispersive long wave equations. *Advances in Water Resources*, 27:1111–1133, 2004.

[10] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI – Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, 2nd edition, 1999.

[11] M. Hamer. Solitary killers. *New Scientist*, August:18–19, 1999.

[12] H. P. Langtangen. *Computational Partial Differential Equations - Numerical Methods and Diffpack Programming*. Texts in Computational Science and Engineering. Springer, 2nd edition, 2003.

[13] G. Pedersen. Three-dimensional wave patterns generated by moving disturbances at transcritical speeds. *J. Fluid Mech.*, 196:39–63, 1988.

[14] G. Pedersen and H. P. Langtangen. Dispersive effects on tsunamis. In *Proceedings of the International Conferance on Tsunamis, Paris, France*, pages 325–340, 1999.

[15] B. F. Smith, P. E. Bjørstad, and W. Gropp. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge University Press, 1996.

[16] S.-B. Woo and P. L.-F. Liu. Finite-element model for modified Boussinesq equations. I: Model development. *J. Waterway, Port, Coastal and Ocean Eng.*, 130(1):1–16, 2004.

Parallel Simulations of Underground Flow in Porous and Fractured Media

A. Beaudoin[a], J.R. De Dreuzy[b], J. Erhel[a] and H. Mustapha[a]

[a]Irisa-Inria, Campus of Beaulieu, 35042 Rennes Cedex, France.

[b]Géosciences Rennes, Campus de Beaulieu, 35042 Rennes cedex, France.

Abstract

In this paper, we present a parallel software for solving linear flow equations in two kinds of subsurface media, a 2D highly heterogeneous porous medium and a 3D fracture network. Parallel computing allows us to solve very large linear systems improving the realism of simulations. For these two applications, we perform a scalability analysis of two parallel solvers : HYPRE and PSPASES. HYPRE is a parallel iterative solver based on a V-cycle multi-grid algorithm. PSPASES is a parallel direct solver based on the Cholesky factorization.

1. Introduction

The prediction of natural underground flow circulation has brought up the concern of medium heterogeneity. Geological heterogeneity in porous media occurs on a large range of scales, that goes from the mineral scale (of the order of the millimeter) to the formation scale (that can be larger than a kilometer). Similarly, rock solid masses are in general fractured and fluids can percolate through networks of interconnected fractures, which are also heterogeneous and multi-scale. Because of the difficulty in reaching the natural medium, the numerical approach seems to be the best solution to study the influence of these two kinds of heterogeneity on the underground flow circulation. The numerical simulations are obtained by performing three main phases : generation of a linear system, solution of the linear system and evaluation of the flow. The first phase is obtained by discretizing the governing equations which are the mass conservation equation and Darcy's law for steady-state, incompressible and single-phase flow in porous media. We use a Mixed Finite Element method to discretize the equations, because it conserves fluxes locally and globally, uses unstructured meshes well-suited to complex geometries and allows heterogeneous and anisotropic permeability tensors. The mesh of the 2D medium is a regular triangular grid, hence Mixed Finite Element method is equivalent to a Finite Difference method. On the other hand, the mesh of the 3D fracture network is rather complex, with interconnected 2D triangular meshes in each fracture. The discrete problem to solve is linear, with a sparse symmetric positive definite matrix. The very strong variability of hydraulic properties leads to an ill-conditioned matrix. For the second phase, we can use direct or iterative methods. The third phase is performed by evaluating Darcy's law. The numerical study of the influence of these two kinds of heterogeneity on the underground flow circulation needs to generate a large number of realistic numerical simulations leading to very large sparse linear systems. In order to reach this objective, we have to overcome two main problems : memory size to generate very large linear systems and run time to solve a large number of linear systems. High performance computing is thus mandatory in this framework. This paper is organized as follows : in Section 2, a description of the parallel algorithms used to generate and solve the large linear systems is given. In Section 3, an analysis of performances of parallel linear solvers applied to two media is presented. Finally, a short summary and our future work are presented in Section 4.

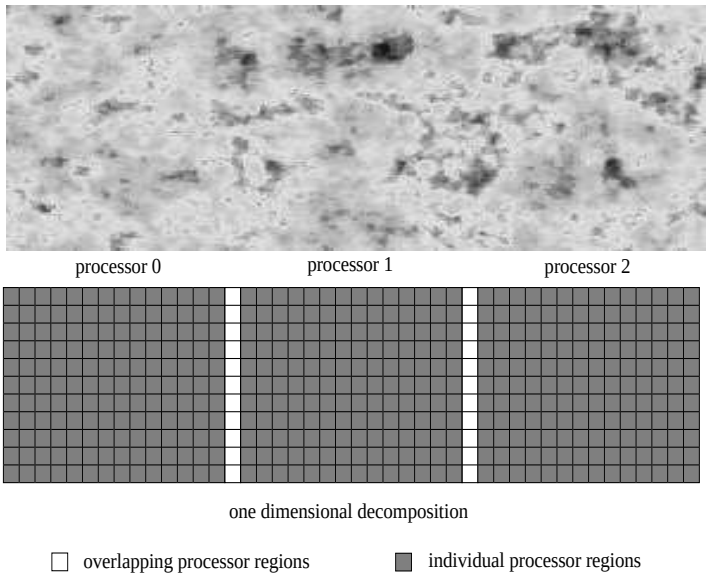


Figure 1. Example of an one-dimensional domain decomposition of 2D highly heterogeneous porous medium (top picture : heterogeneous hydraulic conductivity field and bottom picture : mesh).

2. Parallel Computing

2.1. Parallel matrix generation

In the case of 2D medium, the computational domain is a regular triangular grid on which a random hydraulic conductivity field K is generated. This random hydraulic conductivity field K follows a stationary log-normal probability distribution $Y = \ln(K)$, which is defined by a mean m_Y and a covariance function C_Y. The porous medium is assumed to be isotropic. The covariance function is then defined by $C_Y(\mathbf{r}) = {\sigma_Y}^2 \exp\left(-\frac{|\mathbf{r}|^2}{\lambda_Y}\right)$ where ${\sigma_Y}^2$ is the variance of the log hydraulic conductivity and λ_Y denotes the correlation length scale. To generate the random hydraulic field, a spectral simulation based on the FFT method (Fast Fourier Transform method) is used [11] [10]. The evaluation of Fourier transform in the FFT method has been performed with the FFTW library (Fast Fourier Transform in the West). This FFTW library allows to calculate the Fourier transform on a cluster of processors. Data is then distributed across the processors [4]. The computational grid is divided according to the columns. Thus each processor gets a subset of columns of the computational grid using a one dimensional distribution along the columns. For the flow problem, we have decided to keep this one dimensional decomposition in the y direction. Each processor generates the sub-matrix corresponding to its sub-domain. In order to evaluate the element matrices which are on the boundary of a processor domain, we include one boundary layer of ghost cells that overlaps each sub-domain. These ghost cells are used for temporary storage of grid quantities from neighboring processors. It allows to reduce the communications between the processors during the assembly of linear system (see Figure 1). The 3D fracture network is generated by using a stochastic approach. The network is included in a cube of size L, fractures are ellipses and fracture length is modeled by a random power-law distribution. Eccentricity, orientation and position are also randomly distributed. The density of the network is a parameter of the network generation [2]. The linear system is obtained by discretizing the flow equation on a global mesh of the network. In a first step, all the fracture intersections and boundaries are discretized. A 2D mesh of each fracture is then generated

by using the Emc2 software [6]. It ensures that the intersections are equally discretized in common fractures and that the flux is conserved across the fractures. A similar method is developed in [9], but with non-matching discretized intersections and an adapted nonconforming Mixed Finite Element Method. The equations are then approximated by using the mesh and lead to a linear system of equations. In order to get a symmetric positive definite matrix, a hybrid approach is used [7]. The order of the matrix is the number of edges in the network mesh. The parallel mesh generation relies on a data distribution of fracture structures. In order to get a static balanced task scheduling, we implement a variant of the bin packing algorithm [1]. The mesh generation is embarrassingly parallel, communications occur only to attribute global numbers to mesh edges. Then we infer the data distribution of the mesh and the matrix structures from the parallel mesh generation. Therefore matrix generation is done in parallel with the same distribution. All fracture intersections are processed by one unique processor, which collects matrix data related to intersections from other processors (see Figure 2).

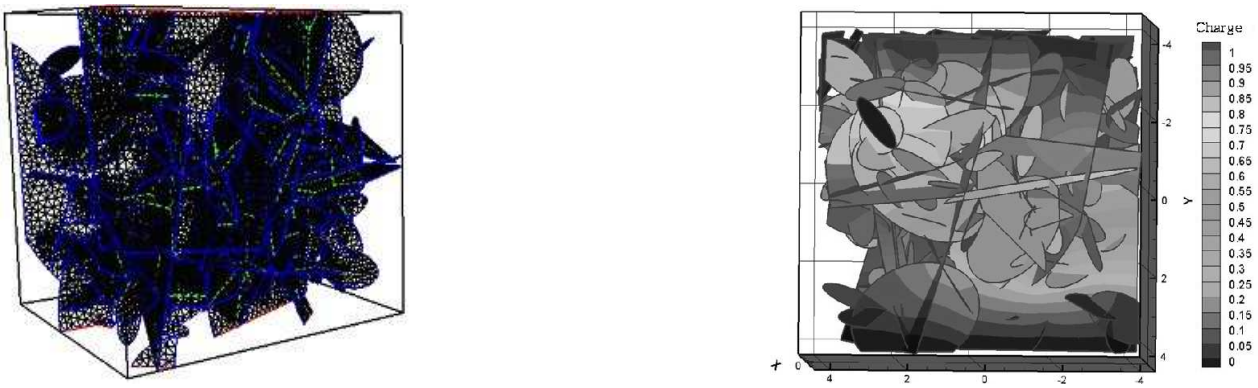


Figure 2. Example of mesh and flow computation of 3D fracture network (left picture : mesh and right picture : flow computation).

2.2. Parallel linear solving

For both 2D highly heterogeneous porous medium and 3D fracture network, the linear system obtained from the discretization of equations with the Mixed Finite Element method is characterized by a sparse symmetric positive definite matrix. For solving these systems, we have investigated both direct and iterative methods. The direct method is based on the Cholesky factorization $A = LL^T$ which is accurate and robust. We use the PSPASES solver, which is an efficient parallel sparse direct solver for symmetric positive definite matrices. Parallelism is based on a distributed-memory paradigm and communications are handled by the MPI library. A slight drawback is that the number of processors must be a power of two, and that there is no sequential version of PSPASES [5]. Because of fill-in in the Cholesky factor L, memory requirements may be a bottleneck for very large linear systems, so it may be necessary to switch to an iterative method. Preconditioned conjugate gradient can be efficient, provided the preconditioner is powerful. For the 2D medium, we have chosen a multi-grid solver which is well adapted to solve linear systems arising from finite difference, finite volume or finite element discretizations of partial differential equations on regular grids. We use the HYPRE library which contains a parallel V-cycle multi-grid algorithm called SMG (Structured Multi-Grid). As the PSPASES library, the HYPRE library uses MPI for the communications

between the processors [3]. Once the linear system has been solved, we compute the hydraulic head and the flux on each element of the computational grid. As for the hydraulic conductivity, the hydraulic head is distributed across all processors. Each processor has the value of hydraulic head on its computational sub-domain.

3. Results and performances

3.1. Tests and architecture

The study of the performance of our parallel software is realized in three tests. In the two first tests, we analyze the complexity and the scalability of the direct solver PSPASES. In the third test, we compare the direct solver PSPASES with the iterative solver HYPRE. This last test is only applied to 2D medium. All tests are performed on a SUN cluster composed of two nodes of 32 computers each. Each computer is a 2.2 Ghz AMD Opteron bi-processor with 2 Go of RAM. Inside each node, computers are interconnected by a Gigabit Ethernet Network Interface, and the two nodes are interconnected by a Gigabit Ethernet switch (CISCO 3750). The characteristic bi-processors of cluster is not used.

3.2. Complexity analysis with a direct solver

The study of complexity is performed by analyzing the CPU time and memory requirements of our two applications. The algorithm is decomposed into three phases: matrix generation, linear solving and flow computation. Figure 3 shows the CPU time of each phase, obtained with two processors, with respect to N, the size of the linear system. We take parallel run times for two processors because the parallel solver PSPASES requires at least two processors. We can observe that the most time consuming phase is the linear solver. Moreover, a complexity analysis shows that the first and third phases have a linear behavior, with a CPU time proportional to the matrix size N. For the fracture network, at least for these experiments, the factorization step is almost linear with N. In contrary, the factorization step has a nonlinear complexity, with a CPU time proportional to N^{α}, where α is about 1.5 for the 2D grid. For both applications, the main memory requirements are to store the matrices A and L. The memory requirements are measured by counting the number of non-zeros in the sparse matrices A and L (in a sparse storage compressed scheme, only nonzero coefficients are stored in 64 bit words). On Figure 4, these two numbers, noted $nz(A)$ and $nz(L)$, are reported for N ranging from $0.5e+06$ to $6.0e+06$. For 2D grids as well as for fracture networks, $nz(A)$ is roughly $5N$, whereas $nz(L)$ is roughly $5Nlog5N$. The analysis of CPU time and memory requirements show that it is necessary to use parallel algorithms to solve large linear systems.

3.3. Scalability analysis with a direct solver

On Figure 5, the total time of a single run for a linear system of size $N = 4.2e + 06$ is reported with respect to the number of processors P ranging from 2 to 64. We can observe that the total runtime is reduced as P increases for both applications. We can also notice that it is sufficient to take 32 processors for the case of 2D medium, whereas for the case of 3D fracture network, we can use effectively 16 processors. In order to understand the parallel performances of PSPASES, we analyze the scalability which refers to the capacity of the algorithm-architecture combination to effectively utilize an increasing number of processors P. The scalability can be evaluated by studying the efficiency E or the speedup S [8]. An algorithm architecture is considered scalable if the efficiency E is fixed when the size of linear system N increases proportionally to the number of processors P. The efficiency E is defined as the ratio of the serial runtime to the parallel runtime. For the linear solving phase, the sequential runtime is unknown because PSPASES uses a number of

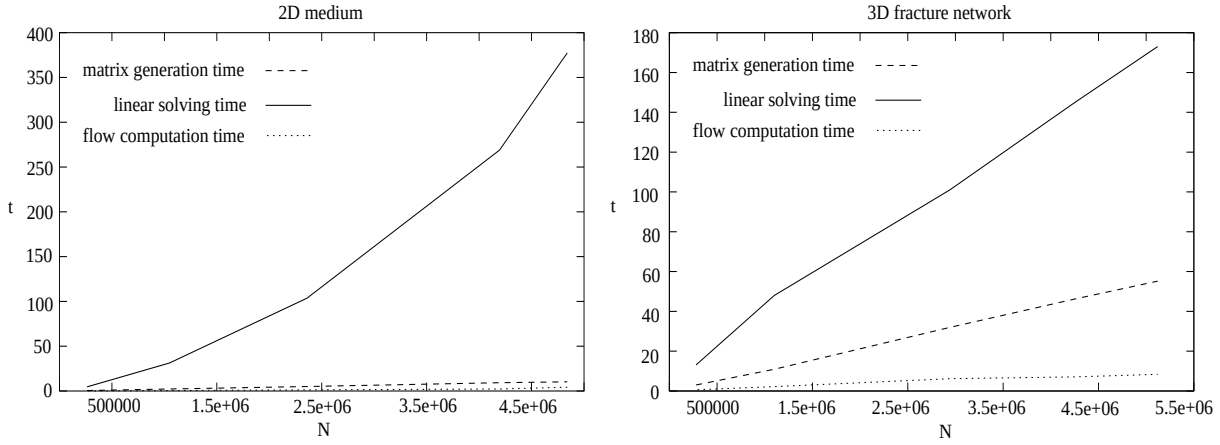


Figure 3. CPU time of three phases (matrix generation, linear solving and flow computation), obtained with two processors, with respect to the size of linear system N for both two applications.

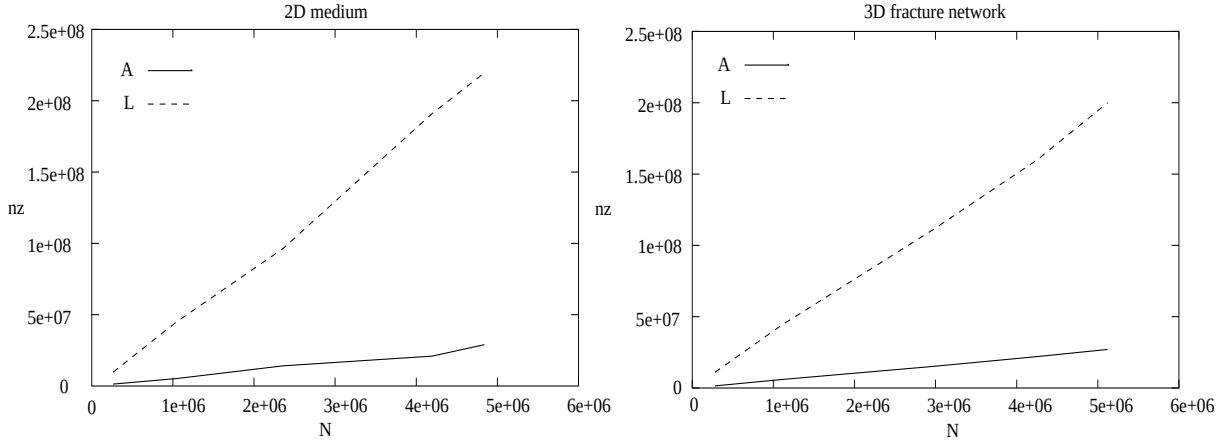


Figure 4. Number of nonzero coefficients nz in the matrices A and L with respect to the size of linear system N for both 2D medium and 3D fracture network.

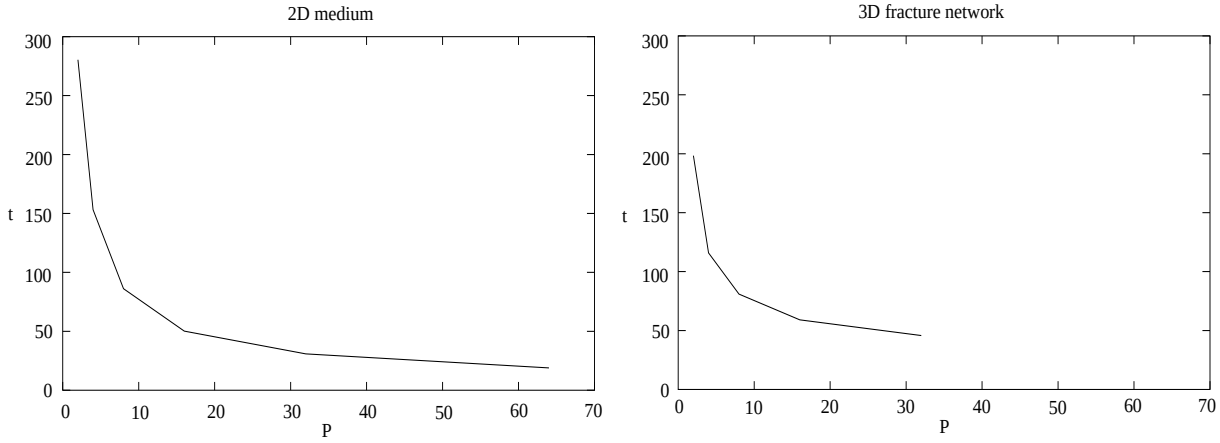


Figure 5. Total run time of a single realization for $N = 4.2e + 06$ with respect to number of processors P for both 2D medium and 3D fracture network.

P	N	T_p	R
2	262144	5.60	11977373
8	1048576	11.33	11844656
32	4194304	25.70	10443374
4	262144	2.92	11502234
16	1048576	6.06	11079774
64	4194304	13.08	10535895

P	N	T_p	R
2	262144	13.10	10006
8	1048576	22.06	5942
32	4194304	38.41	341
4	262144	7.94	16508
16	1048576	16.05	4083
64	4194304	no value	no value

Table 1
Values of the parameter R for various values of (P, N) and for both two applications (left : 2D medium and right : 3D fracture network).

processors which has to be a power of two. But we know that this sequential runtime is proportional to N^α where α is about 1.5 for the 2D grid and 1 for the 3D fracture network. The efficiency is thus proportional to $R = (N^\alpha/(PT_p))$, where T_p denotes the parallel runtime for the linear solving phase. In order to test the scalability of PSPASES, we have evaluated the parameter R by increasing the number of processors P and the size of linear system N with a coefficient equal to 4. The numerical values of R are reported in Table 1 for both applications. From these results, we can conclude that the parallel solver PSPASES is scalable in the problem of 2D medium, as predicted by the theory and observed in [5]. However, it does not appear to be scalable in the problem of 3D fracture network. For the problem of 2D medium, the efficiency E is fixed if the ratio of N to P is maintained constant. Another analysis relies on the speedup S. Since there is no sequential version of PSPASES, the speedup is given by $S = 2T_2/T_P$ where T_2 and T_p are respectively the linear solving times obtained with 2 and P processors. Figure 6 shows the speedup S with respect to the number of processors P for three values of N and for both applications. We can observe that the problem of 2D medium gives values of S larger than those of the problem of 3D fracture network and that the values of S are equivalent for three values of N in the case of 3D fracture network. These two behaviors can be explained by the total parallel overhead T_o which is defined as the total time of the overhead due to parallel processing and is equal to $PT_p - T_s$, where T_s is the serial runtime. The speedup S can then be given by $S = P/(1 + T_o/T_s)$. From [5], we can deduce that the ratio T_o/T_s is proportional to $\sqrt{P/N}$ for 2D medium. This is in good agreement with our numerical values of speedup S. We can also notice that the speedup S decreases at $P = 32$ for $N = 1e + 06$ in the case of 2D medium, probably because the total parallel overhead T_o (including reordering and triangular solving) increases faster than P. An algorithm architecture is considered scalable if the speedup S increases proportionally to the number of processors P when the problem size increases proportionally to P. For the problem of 3D fracture network, the speedup S remains constant when N increases, so that PSPASES is not scalable, probably because the ratio T_o/T_s is independent of N. In this case, parallel computing is used to speedup computations up to eight processors and mainly to increase the memory capacity with more processors.

3.4. Comparison between a direct and an iterative solver

In the case of 2D medium, the parallel direct solver PSPASES gives good performances. But the memory requirements for the matrix L increase faster than the size N of the linear system. The fill-in of matrix L can saturate available memory. As our main objective is to solve large linear systems, we use also a parallel iterative solver, called HYPRE, in order to overcome this difficulty. On Figure 7, the parallel runtime of linear solving phase has been reported with respect to the number P of processors. We can observe that PSPASES gives better performances than HYPRE for

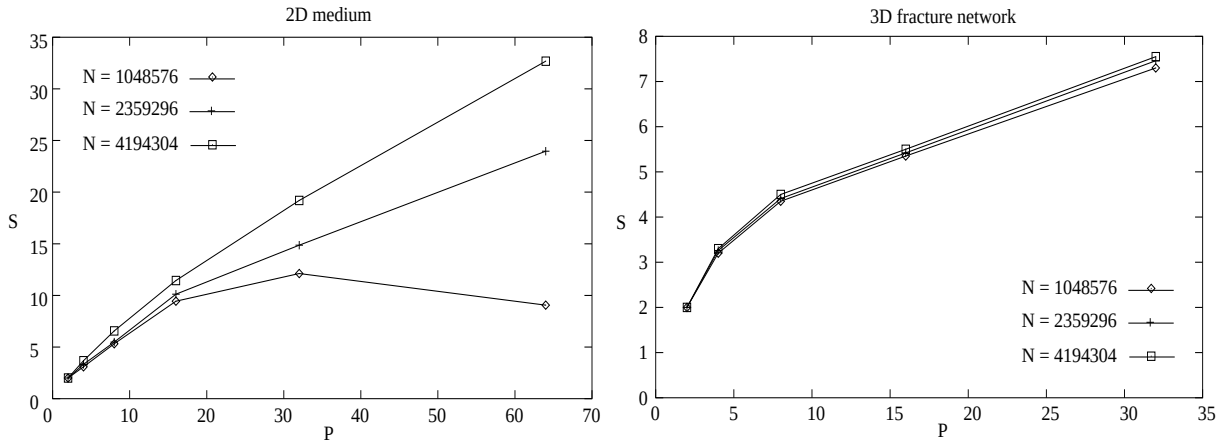


Figure 6. Speed-up S of linear solving with respect to the number of processors P for three values of N and for both 2D medium and 3D fracture network.

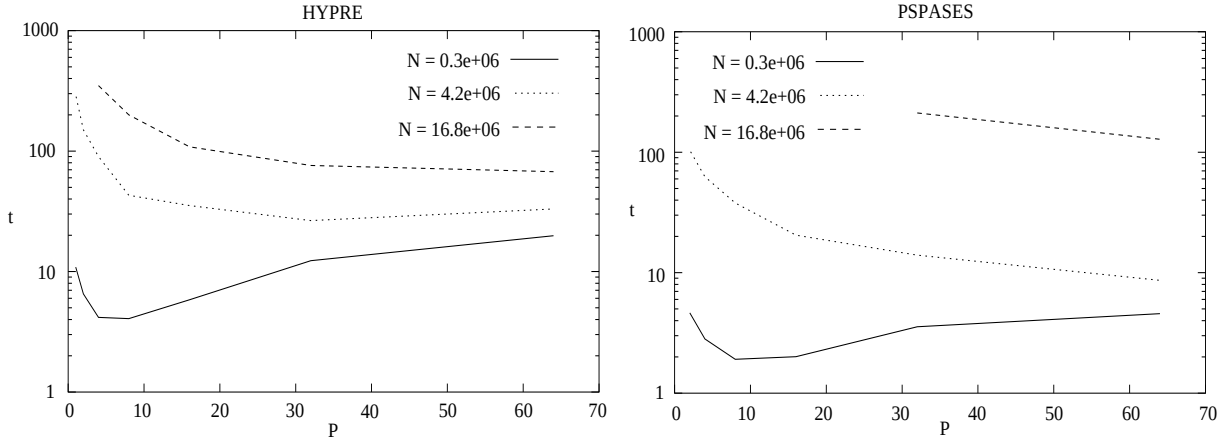


Figure 7. Linear solving time with respect to the number of processors P for N fixed and both PSPASES and HYPRE.

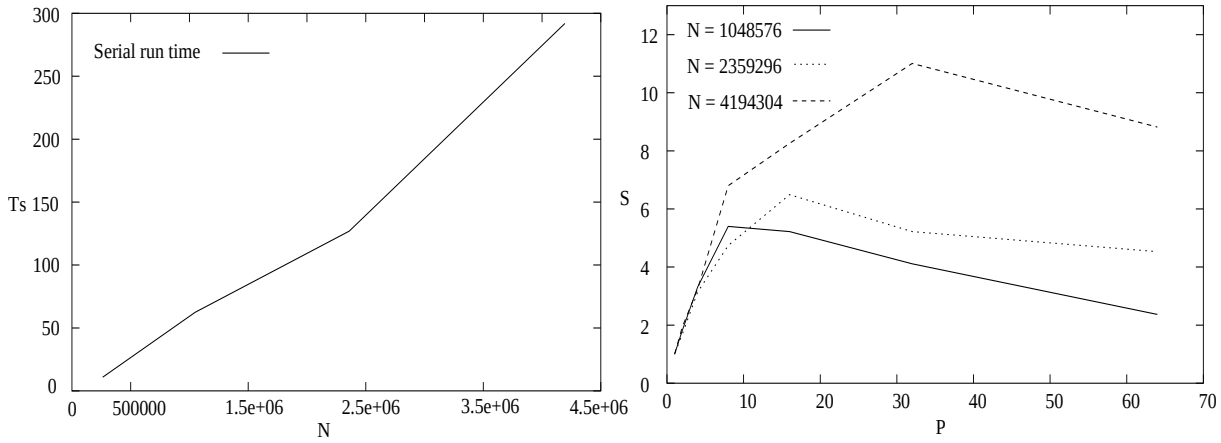


Figure 8. Serial runtime T_s of linear solving phase with respect to the size of linear system N and speedup S with respect to the number of processors P for three values of N obtained with HYPRE.

$N = 0.3e + 06$. For $N = 4.2e + 06$, the two parallel solvers give roughly the same performance. The performance of HYPRE is better than the performance of PSPASES for $N = 16.8e+06$. Figure 8 shows the serial runtime T_s of linear solving phases for values of N ranging from $0.3e + 06$ to $4.5e + 06$ and the speedup S with respect to the number P of processors for three values of N. Speedups are not as important as with PSPASES, but they increase also with N. Therefore we can conclude that HYPRE is somehow scalable for the 2D medium problem. It is necessary to pursue theoretical and practical investigations to measure the isoefficiency function.

4. Summary

In this paper, we have analyzed the performance of a parallel software that solves linear flow equations in 2D porous media or in 3D fracture networks. The scalability study shows that the parallel direct solver PSPASES is scalable in the case of 2D medium, contrary in the case of 3D fracture network. In the 3D fracture network, parallel computing improves the memory capacity, whereas it improves also the linear solving time in the case of 2D medium. However PSPASES can saturate the available memory because of fill-in of matrix L for large linear systems. In order to overcome this problem, the use of parallel iterative solver HYPRE seems to be a good solution. The comparison between the two parallel solvers, shows that HYPRE is efficient for large linear systems. In a future work, we will use HYPRE in the case of 3D fracture network. We will also realize a 3D extension of our parallel software for the case of 2D porous media. Finally, this parallel software will be used in a transport code for simulating the solute migration.

References

[1] S. Albers and M. Mitzenmacher : Average-case analyses of first fit and random fit bin packing. Random structures alg. 16, 240-259, 2000.

[2] J.R. de Dreuzy, P. Davy and O. Bour. Percolation threshold of 3D random ellipses with widely-scattered distributions of eccentricity and size, Physical Review E, vol. 62, 5948-5952, 2000.

[3] R.D. Falgout, J.E. Jones and U.M. Yang : Pursuing scalability for hypre's coceptual interfaces. ACM transactions on mathematical software, 2004.

[4] M. Frigo and S.G. Johnson : The Design and implementation of FFTW3. Proceedings of the IEEE, vol. 93, 216-231, 2005.

[5] A. Gupta, F. Gustavson, M. Joshi, G. Karypis, and V. Kumar. PSPASES : An efficient and scalable parallel sparse direct solver. In parallel numerical computations with applications, T. Yang (ed.), Kluwer international series in engineering and computer science, vol. 515, 1999.

[6] F. Hecht and E. Saltel : Emc2 : Un logiciel d'édition de maillages et de contours bidimensionnels. Rapport technique n 118, INRIA, 1990.

[7] H. Hoteit, J. Erhel, R. Mosé, B. Philippe and P. Ackerer : Numerical reliability for mixed methods applied to fbw problems in porous media. Computational geosciences 6(2): 161-194, 2002.

[8] V. Kumar and A. Gupta : Analyzing scalability of parallel algorithms and architectures. Journal of parallel and distributed computing, 1994.

[9] P.A. Raviart and J. M. Thomas : A mixed hybrid finite element method for the second order elliptic problem. in Lectures Notes in Mathematics 606: 292-315, 1977.

[10] M.G. Trefry, F.P. Ruan and D. McLaughlin : Numerical simulations of preasymptotic transport in heterogeneous porous media: Departures from the Gaussian limit. Water Resources Research, vol. 39, 1063, 2003.

[11] T. Yao : Reproduction of the mean, variance and variogram model in spectral simulation. Mahematical geology, vol. 36, 487-505, 2004.

A parallel software for a saltwater intrusion problem *

É. Canot[a], C. de Dieuleveult[b], J. Erhel[b]

[a]CNRS – IRISA, Campus de Beaulieu, 35042 Rennes

[b]INRIA – IRISA, Campus de Beaulieu, 35042 Rennes

Abstract

In this paper we present a parallel implementation of a 2D numerical model for the solution of a transient density driven flow in porous media. The physical processes involved are multi-scale, therefore computation time are usually long, thus a special effort has been made to speed-up computations by using parallel architectures.

As most of the CPU time is spent in solving large sparse linear systems, it is important to choose an efficient linear solver which can deal with symmetric and non-symmetric sparse matrices. Accordingly, the parallel direct linear solver MUMPS is used for solving both transport and flow. However, scalability is not completely achieved. Therefore, parallelism is generalised to all computations. Matrices and data are distributed thanks to the partitioning of the METIS package instead of centralising on one processor. Actually, each processor treats its own spatial sub-domain and transfers data to the other ones when necessary.

The resulting software is tested on a standard benchmark : the Henry test case. We use a homogeneous parallel cluster of PCs interconnected by a fast network. Investigation on the network and on the MUMPS options are carried out in order to obtain good performances.

1. Introduction

Many areas of the world use groundwater as their main source of freshwater supply. In the particular case of coastal aquifers, one of the major concerns is the seawater intrusion. Moreover, effective management demands a thorough understanding of the variable density groundwater flow system. Numerical simulations help in this sustainable management but they require high performance computing resources.

Generally, transport of solute by groundwater flow does not affect fluid properties, but in the particular case of seawater intrusion it does. The resulting problem becomes difficult to solve because it is highly nonlinear. Indeed transport of saltwater modifies the basic flow by density variations whereas the Darcy velocity calculated by flow, is required to solve advection.

Many numerical models, adapted for this particular case, have been developed : for example, FEFLOW[7], HST3D [13], MOCDENSE [14], SUTRA [17], SWIFT [15] or UG [11].

We present here a parallel program dealing with this topic. The original sequential software TVDV-2D (Transport with Variable Density and Viscosity [2]) has been developed at IMFS in Strasbourg, using robust numerical methods well adapted to density driven flow [18].

Our goal has been to get good performances thanks to parallelism and to allow large scale simulations. Validations of our software are mainly based on a test case, the Henry problem [10] (saltwater front in a confined aquifer initially charged with freshwater).

*This work has been supported by a french government grant, the ACI GRID project called HYDROGRID.

2. Model of saltwater intrusion problem

The model of the TVD2D software is first presented. It is described in [2,18,1]. The governing equations of variable density groundwater flow and solute transport are described in detail by Bear and Bachmat [6].

These equations include, classically, fluid and solute mass balances, generalised Darcy's law and equations of state for the liquid density and viscosity.

2.1. Mathematical model

2.1.1. Fluid equations

The generalised Darcy law can be written as a function of h, the hydraulic head defined by $h = P/\rho_0 g + z$, where P is the pressure, ρ_0 is the density of pure water, g is the gravity acceleration and z is the vertical coordinate. This leads to the following equation :

$$\varepsilon v = -\frac{k\rho_0 g}{\mu}\left(\nabla h + \frac{\rho-\rho_0}{\rho_0}\nabla z\right).$$

where ε is the porosity, v is the fluid velocity, k is the permeability tensor of the porous medium, μ is the dynamic viscosity of the fluid, and ρ is the mass density of the fluid.

Then, conservation of mass gives:

$$\frac{\partial(\varepsilon\rho)}{\partial t} + \nabla.(\rho\varepsilon v) = \rho Q,$$

where Q is a source term.

Most models assume that the effect of temperature can be neglected, porosity is only a function of pressure whereas density and viscosity are functions of pressure and solute mass fraction. Moreover, in our case, we assume that ε and k are constants like fluid viscosity and fluid density is independent of pressure but a linear function of the solute mass fraction.

By defining

$$\alpha = \frac{1}{1-\varepsilon}\frac{\partial\varepsilon}{\partial P},\ \beta = \frac{1}{\rho}\frac{\partial\rho}{\partial P},\ S = \alpha(1-\varepsilon) + \varepsilon\beta,$$

with α the coefficient of compressibility of the porous medium, β the coefficient of compressibility of the fluid and S the specific storativity of the porous medium, the mass conservation law can be written as

$$\rho_0\rho g S\frac{\partial h}{\partial t} + \varepsilon\frac{\partial\rho}{\partial C}\frac{\partial C}{\partial t} + \nabla.(\rho\varepsilon v) = \rho Q.$$

with C is the solute mass fraction.

In most cases, the storage term S is very small or null, therefore the discrete mass matrix can be singular.

2.1.2. Transport equations

The solute transport is governed by a convection-diffusion process. Assuming that $\varepsilon\rho$ is almost constant, the solute mass conservation equation can be written as :

$$\frac{\partial C}{\partial t} + v\nabla C = \nabla.(D\nabla C),$$

where D is the dispersion tensor defined by

$$D = D_c + D_m I,$$
$$D_c = \|v\|(\alpha_L E(v) + \alpha_T(I - E(v))), E_{i,j}(v) = \frac{v_i v_j}{\|v\|^2},$$

where D_m is the molecular diffusion coefficient, α_L is the longitudinal dispersivity, and α_T is the transverse dispersivity.

2.2. Numerical discretisation

The global system is discretised both in space and time. For the transport equation, operator splitting is applied, thus allowing adaptive numerical schemes. The convective part of the transport is spatially discretised by a Discontinuous Finite Element scheme (DFE) stabilised with a slope limiter, whereas the dispersive term in the transport equation is spatially discretised by a Mixed Hybrid Finite Element scheme (MHFE). A MHFE scheme is also used in the flow equations, in order to get an accurate fluid velocity.

After space discretisation, a fully coupled stiff system of differential algebraic equations is obtained, which is discretised in time by an implicit Euler scheme, excepted in the convective part where the mass fraction is explicit in time.

Finally, at each time step, a system of nonlinear equations is solved by using a fixed-point scheme, more precisely a nonlinear Gauss-Seidel iterative method. This allows to separate transport equations from flow equations. The stopping criterion is based on the maximum differences between the heat and the concentration on the edges at iteration k and $k+1$. As far as we know, there is no proof of convergence for this specific nonlinear system but in practice, we get convergence by reducing the time step.

Roughly, each time step can be written :

$$\begin{cases} \rho^{n+1} = Density(C^{n+1}) \\ A_{flow}(\rho^{n+1})h^{n+1} = b_{flow}(\rho^{n+1}, C^{n+1}, h^n) \\ v^{n+1} = Velocity(h^{n+1}) \\ C^* = b_{convection}(v^{n+1}, C^n) \\ D^{n+1} = Dispersion\_Tensor(v^{n+1}) \\ A_{dispersion}(D^{n+1})C^{n+1} = b_{dispersion}(C^*, D^{n+1}) \end{cases}$$

The most time-consuming parts are the linear solvers, involving large sparse matrices $A_{dispersion}$ and A_{flow}. Other parts involve computation of the density, the velocity, the dispersion tensor but also the convection scheme and the matrices calculation. These matrices are computed at each nonlinear iteration since they depend respectively on the velocity and on the density. The matrix $A_{dispersion}$ is symmetric positive definite whereas the matrix A_{flow} is non-symmetric but with a symmetric structure (the non-symmetry comes from the density in the mass balance equation).

3. Parallel implementation

Numerical simulations for saltwater intrusion must deal with a very large number of time steps. Moreover, 3D geometries imply a large number of cells and large linear systems. Therefore, a special effort has been made to accelerate computations.

3.1. Parallel linear solvers

Most of the CPU time is spent in solving large sparse linear systems. Therefore, it is important to choose an efficient linear solver which can deal with symmetric definite positive or non-symmetric sparse matrices in order to reduce execution time.

The choice of the appropriate method is a hard task because of the variety of packages and possible options of each solver [4]. We choose to use a direct method implemented in a parallel library. Our choice is the direct linear solver MUMPS [5] for solving transport and flow.

MUMPS (Multifrontal Massively Parallel Solver) is a package using a multifrontal technique for solving linear systems of equations of type AX=b, where the matrix A is sparse and can be either unsymmetric like in the flow part, symmetric positive definite like in the transport part, or general symmetric. We choose this package because it is free and is known to provide efficient results [9].

The MUMPS solver is decomposed into three steps : symbolic factorisation, numerical factorisation and triangular solvings. The symbolic factorisation is executed only once, in the initialisation step for the flow and in the first iteration step for the transport, because the structure of the matrices is fixed during the whole simulation.

Different options for the pivot order are possible. The pivot order consists in reordering the unknowns of the matrix to reduce fill-in during factorisation.

1. Approximate Minimum Degree (AMD)[3],
2. user-defined ordering,
3. Approximate Minimum Fill (AMF),
4. PORD [16],
5. METIS [12],
6. AMD with automatic quasi-dense row detection (QAMD).

3.2. Data distribution and parallel matrix generation

Scalability is not completely achieved by the choice of a parallel linear solver, because of the other computations.

Therefore, parallelism is generalised to all computations in order to speed up again the execution. This also reduces memory requirements. Indeed, matrices and data are distributed instead of centralising them on one processor. Data are partitioned by the free METIS [12] package according to the elements of the mesh. Indeed, METIS is a well-known package for partitioning large irregular graphs and large meshes, and computing fill-reducing orderings of sparse matrices.

Each subdomain contains the data corresponding to the edges and nodes belonging to these elements. Each processor treats its own sub-domain and transfers data to other interfaces when necessary.

Currently, the matrix needed by MUMPS is distributed on processors but neither the right-hand side nor the solution which are centralised on processor 0. It is one restriction of MUMPS, but it is planned to be changed in the next version of MUMPS. Otherwise, only the I/O operations are centralised on a single processor.

4. Results

4.1. Test case

In order to validate our modifications, a classical test case is used : the Henry [10] problem.

This seawater intrusion problem describes the advance of a saltwater front in a confined aquifer initially charged with freshwater.

This test case admits a steady state which is represented in Figure 1. The water is considered saturated in salt ($c = 1$) when the density is $1200\ kg/m^{\ 3}$ whereas the density of fresh water is equal to $1000\ kg/m^{\ 3}$.

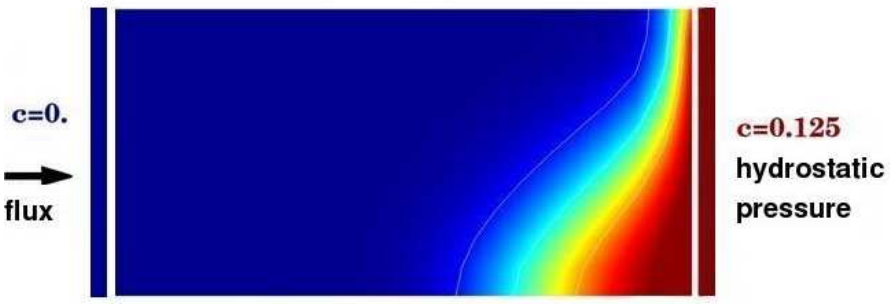


Figure 1. Advance of a saltwater front in a confined aquifer initially charged with fresh water : Henry test case at the steady state and boundary conditions of the problem.

Table 1
Parameters for the Henry problem.

Permeability	$kx = ky = 1.0204 \times 10^{-9}\ m^2$
Porosity	$\epsilon = 0.35$
Dispersivity	$\alpha_L = \alpha_T = 0\ m$
Molecular diffusion coefficient	$D_m = 18.86 \times 10^{-6}\ m^2 s^{-1}$
Flux	$Q = 6.6 \times 10^{-5}\ kg/s$
State equations	$\rho = \rho_0 + 200 C_m$ (C_m is the mass fraction)
	$\mu = 10^{-3}\ Pa.s$
Domain	$2 \times 1\ m$

Here, the density variations are small, but this test case is the first stage of the model validation because of the existence of a semi-analytic steady-state solution. It has become a classic test of variable-density flow. The steady state appears after 120 minutes. With this problem, the results obtained with two regular meshes, one with 254 horizontals elements by 126 verticals elements and the other with 510 by 254 elements, are presented.

We have also done experiments with the Elder test case [8], which deals with tens of years but is more physically unstable.

4.2. Choice of the network

Parallel experiments have been firstly run using MPI on a Fast Ethernet (100 Mb/s) network with bi-processors Intel Xeon (CPU 2.4GHz, cache 512KB). But the results are not so good. In fact, the matrix concerned are relatively small with little time of calculation and a very sparse structure. The use of a high latency network in connection with relatively small volume of data exchanged explains these poor results.

Thus, similar machines are used but with a faster network, Myrinet (2Gb/s) to obtain efficient results.

4.3. Choice of the options

As the meshes used are regular, we investigated if a nested dissection on a regular grid could improve performances. We have defined an ordering based on the following idea :

It consists in partitioning recursively in two parts the original mesh in the largest dimension, and then, renumbering the edges accordingly to this partition. The mesh ordering is then optimized as

we can see on Figure 2 with the representation of the sparsity structure of the matrix.

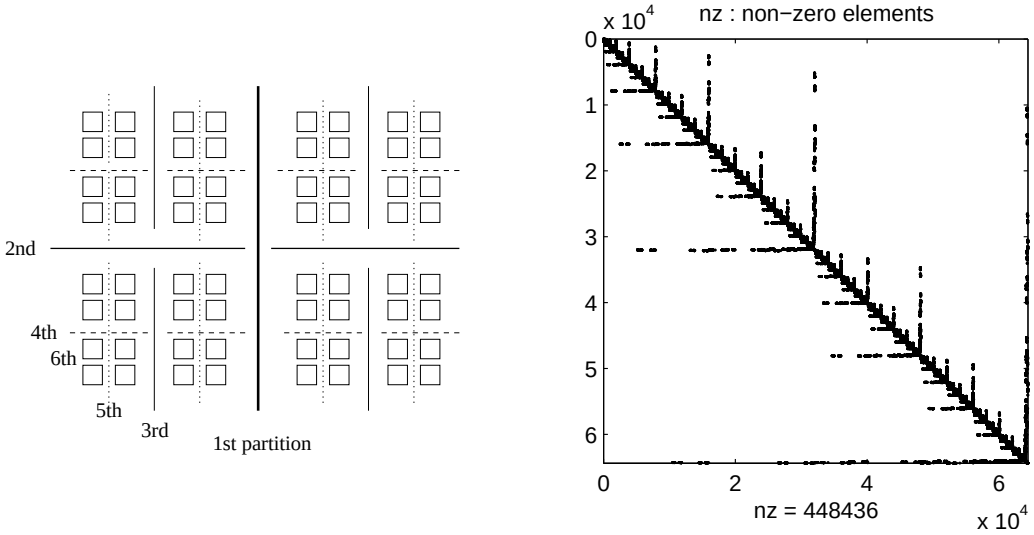


Figure 2. Renumbering of the edges, the partitioning and matrix sparsity structure for a mesh of 254x126 elements.

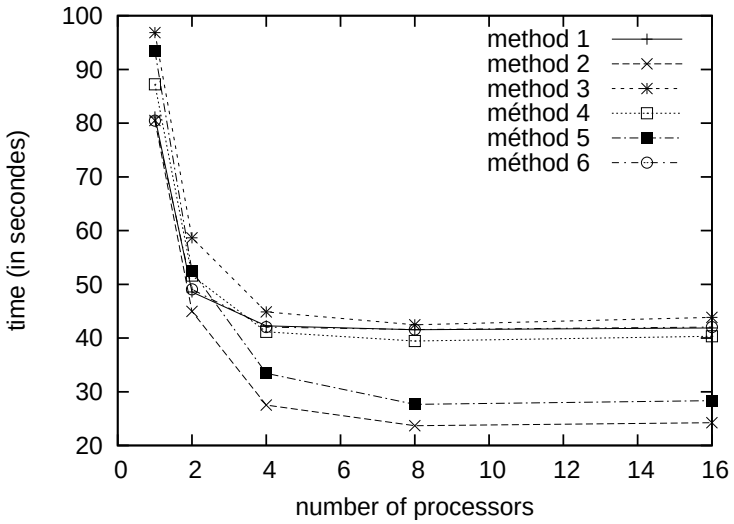


Figure 3. Time spent in MUMPS for different methods of pivot order with mesh 254x126 elements for the Henry test case and 10 time steps.

We see on Figure 3 that the best pivot order is with our user-defined ordering (method 2) just before METIS one (method 5). Actually, these two methods share the same principle, nested dissection, but whereas method 2 is applied only on regular meshes, method 5 is applied on any mesh. Because of this difference, method 5 is used in the following studies.

4.4. Parallel results

Time measurements are reported in Figure 4. In this figure, three different parts of the computation are plotted :

- the MUMPS parallel solving timing,
- the initialisation, the storage and the visualisation timings, which are sequential and are not representative of the calculation. Indeed, the initialisation only occurs one time whereas the storage and the visualisation are optional,
- the other parts of the calculation which are parallelised thanks to the METIS partitioning.

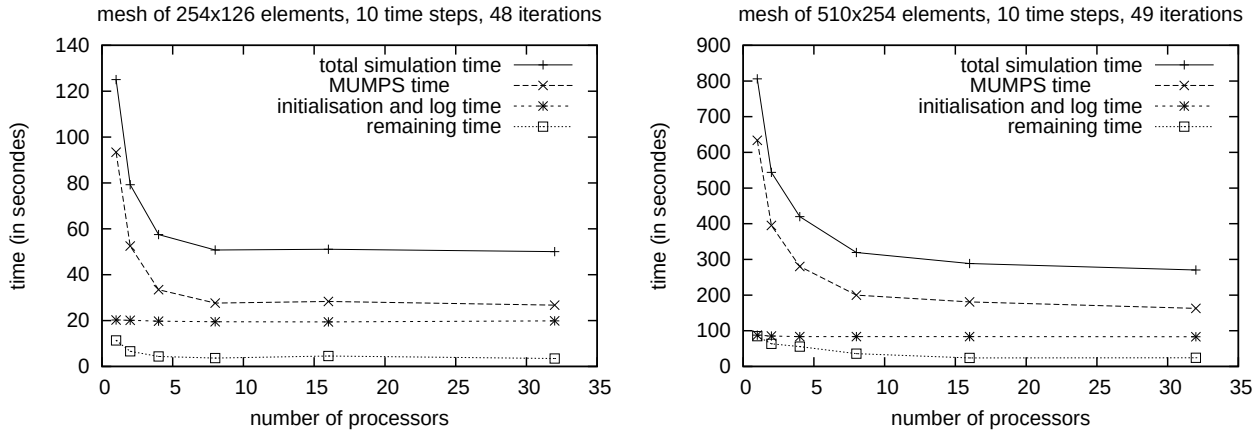


Figure 4. time results with MUMPS and the Henry problem on 2D meshes.

As expected, solving the linear systems takes a large part of the total simulation time. For the smallest mesh (system size N= 64262), parallelism is very efficient up to 4 processors. For the largest mesh (system size N=259590) performances are still good with 8 and 16 processors.

5. Conclusion

Numerical simulations in hydrogeology are quite often based on coupled models like the saltwater intrusion problem involving strongly flow and transport coupling.

Our parallel software allows to speed up the execution especially by the use of the parallel linear solver but also by data partitioning.

Moreover, simulations with 3D geometry should show better performances because the matrices would be much larger. Besides, parallelisation of the visualisation could also be investigated as well as the comparison with other linear solvers. Another direction of work will be to improve the coupling of the flow and the transport.

References

[1] P. Ackerer, A. Younes, S.E. Oswald, and W. Kinzelbach. On modelling of density driven fbw. *Calibration and Reliability in Groundwater Modelling : Coping with Uncertainty (Red book of the ModelCARE 99 Conference), IAHS Publication*, 265:377–384, 2000.

[2] Ph. Ackerer, A. Younes, and R. Mosé. Modeling variable density flow and solute transport in porous medium : 1. numerical model and verification. *Transport in Porous Media*, 35:345–373, 1999.

[3] P. Amestoy, T. A. Davis, and I. S. Duff. An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications*, 17:886–905, 1996.

[4] P. Amestoy, I.S. Duff, J.-Y. L'Excellent, and X. S. Li. Analysis and comparison of two general sparse solvers for distributed memory computers. *ACM Transactions on Mathematical Software*, 27(4):388–421, 2001.

[5] P. R. Amestroy, I. S. Duff, J.-Y. L'Excellent, and J. Koster. *MUltifrontal Massively Parallel Solver (MUMPS Version 4.3) Users' guide*, July 2003.

[6] J. Bear and Y. Bachmat. *Introduction to Modeling of Transport Phenomena in Porous Media*. Kluwer Academic Publishers, Dordrecht, 1991.

[7] H. J. Diersch and O. Kolditz. Coupled groundwater flow and transport: 2. thermohaline and 3d convection systems. *Advances Water Ressource*, 21(5):401–425, 1998.

[8] J. W. Elder. Numerical experiments with a free convection in a vertical slot. *Journal of Fluid Mechanics*, 24:823–843, 1966.

[9] A. Gupta. Recent advances in direct methods for solving unsymmetric sparse systems of linear equations. *ACM Transactions on Mathematical Software*, 28(3):301–324, 2002.

[10] H. R. Henry. Interfaces between salt water and fresh water in coastal aquifers. Technical Report 1613-C, U.S. Geological Survey, Water Supply Paper, 1964.

[11] K. Johannsen, W. Kinzelbach, S.E. Oswald, and G. Wittum. The saltpool benchmark problem - numerical simulation of saltwater upconing in a porous medium. *Advances in Water Resources*, 25(3):335–348, 2002.

[12] George Karypis and Vipin Kumar. Metis a software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing ordering of sparse matrices version 4.0. Technical Report MN 55455, University of Minnesota, Department of Computer Science/Army HPC Research Center, September 1998.

[13] K. L. Kipp. Guide to the revised heat and solute transport simulator : Hst3d - version 2. Technical Report 97-4157, US Geological Survey, Water Resources Investigations, 1997.

[14] L. F. Konikow, P.J. Campbell, and W. E. Sanford. Modelling brine transport in a porous medium : a re-evaluation of the hydrocoin level 1, case 5 problem. In K. Kodar and P. Van Der Heijde, editors, *Calibration and Reliability in Groundwater Modeling*, 237, pages 363–372. IAHS Press IAHS Publication, 1996.

[15] M. Reeves, D. S. Ward, N. D. Johns, and R. M. Cranwell. Theory and implementation of SWIFT II, the sandia waste-isolation flow and transport model for fractured media. Technical Report SAND83-1159, Sandia National Laboratory, 1986.

[16] J. Schulze. Towards a tighter coupling of bottom-up and top-down sparse matrix ordering methods. *BIT*, 4(800-841):2001, 41.

[17] C. I. Voss. A finite element simulation model for saturated-unsaturated fluid-density dependent groundwater flow with energy transport or chemically-reactive single species solute transport. Technical Report 84-4369, US Geological Survey, Water Resources Investigations, 1984.

[18] A. Younès, Ph. Ackerer, and R. Mosé. Modelling variable density flow and solute transport in porous medium : 2. re-evaluation of the salt dome flow problem. *Transport in Porous Media*, 35:375–394, 1999.

A High-Performance Parallel Device Simulator for High Electron Mobility Transistors

N. Seoane [a], A. J. García–Loureiro [a], K. Kalna [b] and A. Asenov [b]

[a]Department of Electronics and Computer Engineering, Univ. Santiago de Compostela, 15782 Santiago de Compostela, Spain. e-mail: natalia@dec.usc.es

[b]Device Modelling Group, Dept. Electronics & Electrical Engineering, University of Glasgow, G12 8LT Glasgow, United Kingdom.

Abstract

Three–dimensional simulators are nowadays essential in semiconductor device simulation in order to study fluctuation effects when devices are scaled to gate lengths approaching nanometre dimensions. To take into account these effects it is necessary to perform statistical studies of atomistic simulations, which have a high computational cost, being essential its minimization. In this work we carry out an analysis of the parallel performance of a 3D device simulator for HEMTs based on the drift–diffusion approximation. We also analyse the convenience of reusing the ILU factorisations in order to minimize execution times. Numerical results show superlinear efficiency values up to 32 processors in the resolution of the Poisson equation, and a lowering of the performance with the increase of the number of processors in the solution of the electron continuity equation. The results were obtained in a Cluster HP Integrity Superdome.

1. Introduction

High Electron Mobility Transistors (HEMTs) [1] are being scaled to gate lengths approaching nanometre dimensions. At these scales, the influence of several sources of fluctuations in doping and material composition may significantly degrade the reliability and performance of the devices. In this case 2D models, which neglect the depth of the device, can not be used to take into account these fluctuations effects and have to be replaced by 3D simulations.

To study the impact of fluctuations it is necessary to perform statistical analysis, which increase the computational cost. Standard workstations are not well suited to carry out the large number of simulations required to get statistically significant results keeping a reasonable execution time. To overcome this problem, parallel computers have to be employed in order to speed–up the whole simulation process.

In this work, we describe the design and investigate the parallel performance of a 3D parallel device simulator [2] for HEMTs, based on the drift-diffusion (D–D) approach to the semiconductor transport. The D–D approach constitutes a system of coupled, nonlinear partial differential equations. Finite element methods have been applied to discretise these equations by using tetrahedral elements. Domain decomposition methods have been used to solve the linear systems arising from the linearisation of the D–D equations. The 3D simulator has been developed for multicomputers using a Multiple Instruction Multiple Data strategy (MIMD) under the Single Program Multiple Data paradigm (SPMD) and the Message Passing Interface (MPI) standard library.

The paper is organised as follows. Section 2 briefly presents the mathematical expressions of the D–D transport model. Section 3 describes the numerical techniques used in the simulation process. Results obtained are presented in Section 4 while conclusions are drawn up in Section 5.

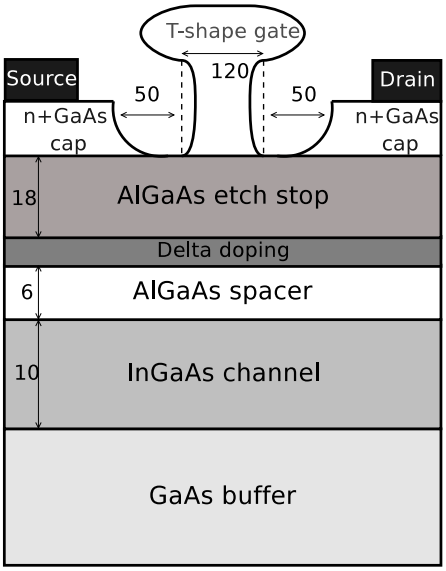


Figure 1. Cross–section of the 120nm gate length pHEMT structure

2. Physical Model

Our 3D parallel device simulator is based on the D–D transport model. The drift–diffusion equations of this model are a system of three coupled, nonlinear equations which describe the relation between the electrostatic potential and the densities of the charge carriers in a semiconductor device. The equations of this model are Poisson equation and electron and hole continuity equations. In stationary state they can be written in the following form:

$$\begin{cases} Find\ (\phi, \phi_n, \phi_p)\ so\ that \\ -div(\epsilon\nabla\phi) + q[n(\phi,\phi_n) - p(\phi,\phi_p) - N_D^+ + N_A^-] = 0 \\ -div(q\mu_n n(\phi,\phi_n)\nabla\phi_n) + qGR(\phi,\phi_n,\phi_p) = 0 \\ -div(q\mu_p p(\phi,\phi_p)\nabla\phi_p) - qGR(\phi,\phi_n,\phi_p) = 0 \\ with\ mixed\ Dirichlet--Neumann\ boundary\ conditions. \end{cases} \quad (1)$$

The unknowns of the problem are ϕ, the electrostatic potential, ϕ_n, the quasi–Fermi level for the electrons and ϕ_p, the quasi–Fermi level for the holes. The electron charge is denoted by q. The mobilities of the electrons and the holes are denoted by μ_n and μ_p respectively and are material dependent. GR is a function which represents the total recombination term. This function may have different expressions depending on the physics taken into account. N_D^+ and N_A^- are the doping effective concentration. The concentration in electrons and holes are n and p.

We have implemented a specific formulation to accelerate the simulation time for HEMTs, due to they are n–type majority carrier devices. Therefore, far from a breakdown we can neglect the hole continuity equation and solve only the Poisson and the electron continuity equation.

3. Three–Dimensional Drift–Diffusion Simulation

The solution scheme is based on the decoupling of the nonlinear Poisson and electron continuity equations in an iterative process. These two equations are discretised using the finite element method (FEM). We have used an unstructured tetrahedral mesh where we have placed more nodes near the

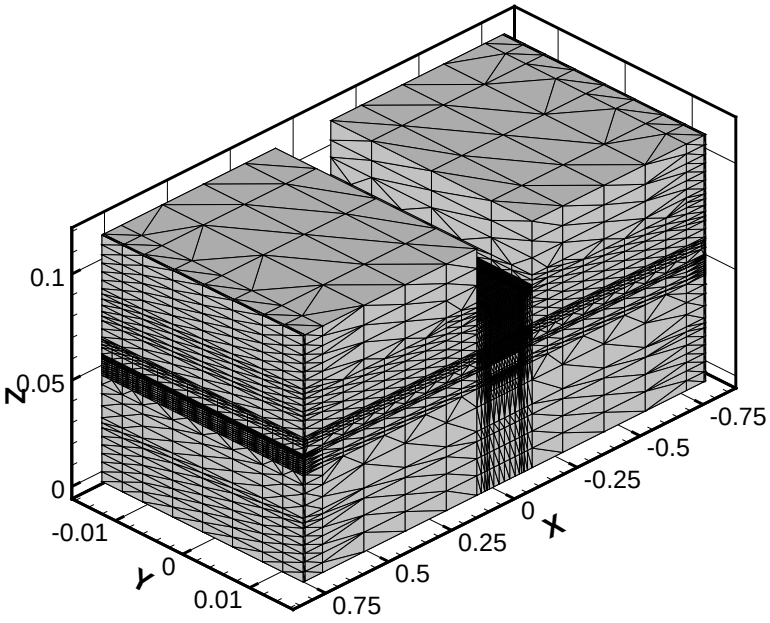


Figure 2. Tetrahedral mesh of the HEMT device generated by the MMG program

interface between different areas of the device, because it is in these areas where we have the greatest gradients of the unknowns of the problem. Then, the METIS program [3] is used to partition the mesh. In this way, the mesh is divided into sub–domains such that we have one for each processor. The same program was subsequently used to relabel the nodes in the sub–domains with the purpose of obtaining a more suitable rearrangement.

The discretisation of the equations leads to two nonlinear systems of algebraic equations. Each of these nonlinear systems is solved by a Newton–Raphson iterative method [4]. Moreover, at every step of the Newton method, a particular linear system has to be solved. The linear system is sparse, badly scaled and ill–conditioned due to the high dynamics of the quantities involved in the simulation and the lack of diagonal dominance in the case of the electron continuity equation [5].

To solve the linear systems of equations we have employed the PSPARSLIB library [6]. This library solves sparse linear systems which are distributed over processors. It uses domain decomposition preconditioners, such as Additive Schwarz, Multicolor SOR and Schur complement methods.

Domain decomposition methods refer to a collection of techniques based on the principle of divide and conquer. If we consider the problem of solving an equation on a domain Ω partitioned in p subdomains Ω_i, such that

$$\Omega = \bigcup_{i=1}^{p} \Omega_i \tag{2}$$

domain decomposition methods attempt to solve the problem on the entire domain by a problem solution on each local subdomain Ω_i.

An analysis of the resolution methods and preconditioning techniques employed in the PSPARSLIB library has been done [7], and the lowest execution times were obtained with the Additive Schwarz method. This algorithm is similar to a block–Jacobi iteration and consists of updating all the new

Table 1
General information about the meshes used in the simulation

Name	Nodes	Tetrahedrons	NNZ	Mesher
S1	26,726	144,608	380,672	MMG
S2	29,012	147,682	398,102	QMG
M1	76,446	433,824	1,116,664	MMG
L1	221,760	1,253,760	3,223,110	MMG

components from the same residual. The basic additive Schwarz iteration would therefore be as follows:

1. Obtain $y_{i,ext}$
2. Compute local residual $r_i = (b - Ax)_i$
3. Solve the local linear system $A_i\delta_i = r_i$
4. Update solution $x_i = x_i + \delta_i$

where $y_{i,ext}$ are the external interface nodes.

To solve the linear system $A_i\delta_i = r_i$ a standard Incomplete LU factorisation with Threshold (ILUT) preconditioner combined with Flexible Generalized Minimal Residual method (FGMRES) for the solver associated with the blocks is used [8]. This is a right–preconditioner variant of the GMRES method that allows the preconditioner to vary at each step. Some zeros in the original matrix may well become nonzeros during the course of ILUT factorisation. The number of the new nonzero elements is indicated with the defined fill–in parameter.

One factor which can affect the convergence of the linear system is the tolerance used for the inner solver. As accuracy increases, the number of outer steps may decrease. However, since the cost of each inner solver increases, this often offsets any gains made from the reduction in the number of outer steps to achieve convergence. It is interesting to observe that the required communication, as well as the overall structure of the routine, is identical with that of matrix–vector products.

4. Numerical Results

The numerical results have been obtained in an HP Superdome Cluster [9] formed by two HP Integrity Superdome servers, each with 64 Itanium2 1.5 GHz, 6 MB cache processors. The main memory of the system is 384 Gbytes and the theoretical peak performance is 768 Gflops.

The 3D device drift–diffusion simulator has been applied to study of a 120nm pHEMT. Simulated characteristics have been compared to data obtained for the 120nm gate length pHEMT designed and fabricated by the Nanoelectronics Research Centre at the University of Glasgow [10]. The schematic cross-section of the simulated device is shown in Figure 1.

The meshing in the 3D simulator is carried out using two programs, the QMG [11] and the MMG [12]. An example of a tetrahedral mesh arising from the MMG program is shown in Figure 2. To accomplish our study we employ four meshes with different size. Their main characteristics are shown in Table 1.

The study has been divided in two sections. In the first one we present, for the Poisson equation in equilibrium, an analysis of the convenience when reusing the ILU factorisations which are used

Table 2
Solving times for Poisson equation in equilibrium reusing the ILU factorisation

Mesh	Processors	$t_{no\_reusing}$(s)	$t_{reusing\_1\_iter}$(s)	$t_{reusing\_2\_iter}$(s)	$t_{only\_a\_first\_ILU}$(s)
S1	1	2250	2200	2174	2150
	2	848	770	783	1013
	4	269	254	248	287
	8	96	80	77	81
	16	38	31	30	32
M1	2	7206	5544	6262	9042
	4	2000	1841	1848	1849
	8	697	655	633	628
	16	214	189	170	181
	32	70	60	57	65
L1	8	5543	4920	6560	4379
	16	1634	1609	1678	1638
	32	591	571	609	746

as preconditioners for Newton iterations and a study of the parallel efficiency. In the second section we show the parallel efficiency of the complete device code.

For this purpose, we have employed the standard definition of the efficiency

$$E(p) = \frac{t_1}{t_p\, p} \tag{3}$$

where t_1 and t_p are the times to execute the workload on a single processor or on p processors respectively.

4.1. Parallel Efficiency of the Poisson Equation in Equilibrium

The first part of this work is related to the evaluation of the ILU factorisations reuse on the parallel performance. ILU factorisations are used as preconditioners for the linear systems with the aim of minimise the execution time. To solve the linear systems we employ the Additive Schwarz domain decomposition method and implement it in each subdomain of the FGMRES solver. FGMRES is preconditioned by the incomplete LU factorisations depending on a particular level of fill–in. In this case, the *fill* parameter used to obtain our results is 70, being $2 \cdot fill$ the maximum number of fill–in elements per row that can be introduced in the structure of outgoing data. Although the matrices change during the Newton iterations, it is possible to reuse same factorisations as an attempt to minimise the cost of solving the linear systems.

The reuse of factorisations slightly improves the performance. Table 2 illustrates that this is more important in the sequential case, where the execution time for Poisson equation always decreases with the increase in the reuse of the same ILU factorisation. However, with the increase of the number of processors employed, reusing always an initial ILU becomes less efficient and we obtain the lowest execution time when we reuse the same ILU one or two iterations.

Figures 3 and 4 show the parallel efficiency for the solution of the Poisson equation in equilibrium for the meshes S1 and M1. Similar results were obtained using the mesh L1. It is also shown the influence of different ILU factorisation reusing conditions for comparative purpose. As we can see we have obtained a surprisingly high superlinearity behaviour. Moreover, in all the studied cases, the parallel efficiency increases with the number of processors employed. The relative increase

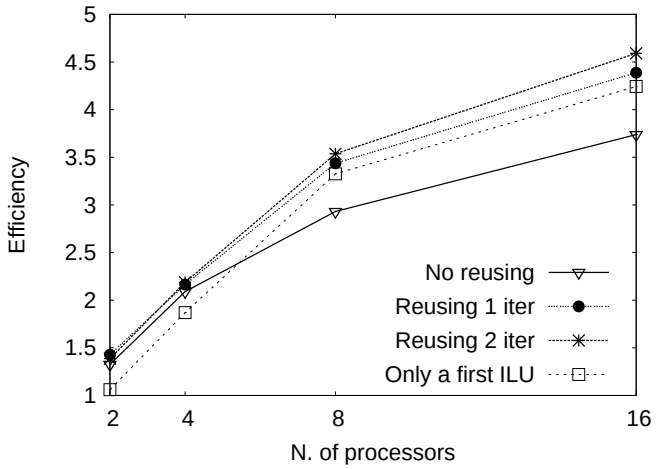


Figure 3. Parallel efficiency for the solution of the Poisson equation in equilibrium using the S1 mesh

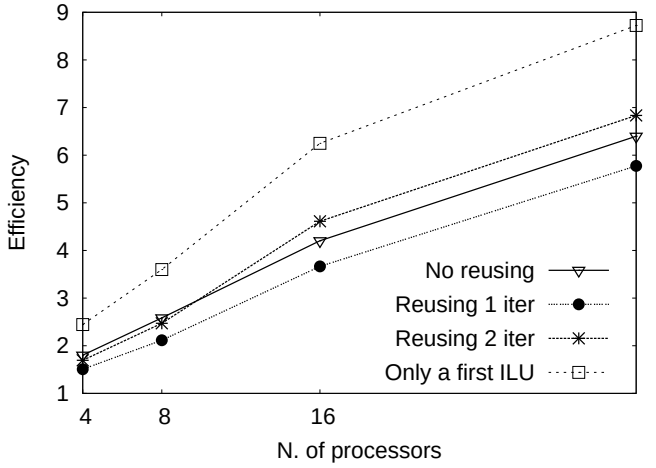


Figure 4. Parallel efficiency for the solution of the Poisson equation in equilibrium using the M1 mesh

is approximately constant, being on average about 1.4 when doubling the number of processors, provided that the size of the subdomains is not too small.

To explain the superlinearity we consider the influence of three factors. First, we take into account cache effects. As the number of processors increase, the number of cache misses decrease. More processors result in a smaller mesh partition size, therefore it fits better in the cache. The higher cache hit rate also results in fewer memory conflicts. The subsequent reduction in memory access times contributes to the parallel efficiency [13]. Second, we consider the influence of the increase of the number of processors in the iterative method, since methods based on domain decomposition are highly parallel [14]. As we stated above, to solve the local nodes within each subdomain we have employed a FGMRES iterative algorithm. Different preconditioning techniques can be applied to this method, we have tried both the PGMRES iterative method and the ILU preconditioner. PGMRES procedure is a simple version of the ILUT preconditioned GMRES algorithm. Although we have found a slight increase in the number of FGMRES iterations when we increase the number

Table 3
Solving time and efficiency for the solution of the Poisson equation in equilibrium and the complete simulation for the S2 mesh

Processors	Poisson time(s)	Poisson efficiency	Complete sim. time(s)	Complete sim. efficiency
1	2698		6579	
2	1099	1.228	2839	1.158
4	428	1.576	1593	1.032
6	297	1.511	939	1.167
16	65	2.580	634	0.648

of processors employed, the cost of each iteration noticeably decreases. And finally, we have to take into account ILU factorisations. The reduction in the size of the matrix is not linear with the increase of the number of processors, therefore the lowering in the factorisation time is higher than the increase in the number of processors.

4.2. Parallel Efficiency of the Complete 3D Simulator

The efficiency analysis of the 3D parallel simulator has been divided in two parts. First, we have solved the Poisson equation in equilibrium. Then, we have obtained the complete simulation time for one point on the I-V curve. In this case the contribution of the Poisson and electron continuity equations are considered. Due to the electron continuity equation properties it is necessary to increase the fill–in in order to achieve the convergence of the system. Therefore the *fill* parameter used to obtain our performance results is 700.

The obtained results for the S2 mesh are summarised in Table 3. The second two columns in this table illustrate time and efficiency for the solution of the Poisson equation in equilibrium. This task is very well parallelizable which can be seen from the superlinear efficiency for up to 16 processors used in this investigation. The execution times are higher than the ones obtained in the previous section because of the different value of fill–in employed.

The complete simulation time and efficiency for a one point on I-V characteristics are also shown in the last two columns of Table 3. The behaviour of the complete simulation for the one point is different because of the influence of the electron continuity equation. In this case, the efficiency drops with the increase of the number of processors. In domain decomposition methods, the partition of the mesh into a higher number of subdomains leads to a reduction of the internal nodes, which causes an increase both in the communications and in the computational effort, mainly due to the higher number of iterations of the inner solver and therefore in the matrix–by–vector multiplications, which are usually the most time consuming part of the iterative solver. This increase in the number of iterations is much more noticeable when we are solving the electron continuity equation, causing a decrease in the parallel performance.

5. Conclusions

3D parallel simulations are essential tools in order to study effects of fluctuations, both in doping and in material composition, when semiconductor devices are scaled into deep submicron dimensions. In this work we have developed a high performance parallel devices simulator for High Electron Mobility Transistors (HEMTs).

The objectives of this paper were twofold, first to analyse the convenience of reusing the ILU factorisations for the Poisson equation used as preconditioners in order to minimize execution times. Second, to study the parallel performance of a 3D parallel semiconductor device simulator, based on the drift–diffusion approximation to the semiconductor transport.

The obtained results indicate that the reuse of the ILU factorisations slightly improves the performance, obtaining the lowest execution times when we reuse the factorisations one or two iterations. With respect to the parallel efficiency study, the resolution of the electron continuity equation is the bottle–neck of the simulation, limiting the scalability and performance of the simulation. On the other hand, the solution of the Poisson equation obtains high parallel efficiency, presenting superlinear behaviour in all the studied cases.

Open questions remain in this study. The results have shown that the main limitation to the performance of the 3D simulator is the solution of the electron continuity equation, so that it should be very interesting to apply a more suitable resolution method and preconditioning technique to solve the linear systems associated with the electron continuity equation.

Acknowledgements.

This work was partly supported by the Spanish Government (CICYT) under the project TIN2004-07797-C02. It has also been performed under the Project HPC–EUROPA (RII3–CT–2003–506079), with the support of the European Community – Research Infrastructure Action under the FP6 Structuring the European Research Area Programme. We are particular grateful to CESGA (Galician Supercomputing Center) and Carlos Fernández for providing access to the HP Superdome system.

References

[1] P. Roblin and H. Rohdin, "High–speed heterostructure devices", Cambridge University Press (2002).
[2] A. García Loureiro, K. Kalna and A. Asenov, "3D Parallel Simulations of Fluctuation Effects in pHEMTs", *J. Comput. Electron.* **2**, 369-373 (2003).
[3] G. Karypis and V. Kumar, "METIS: A software package for partitioning unstructured graphs", University of Minnesota, 1997.
[4] R.E. Bank and D.J. Rose, "Global approximate Newton Methods", *Numerische Matematik*, **37**, 279-295 (1981).
[5] S. Rollin, O. Schenk and A. Gupta, "The effects of unsymmetric matrix permutations and scalings in semiconductor device and circuit simulation", *IEEE Trans. CAD Integ. Circ. Systems*, **23** (2004).
[6] Y. Saad, Gen-Ching Lo, and S. Kuznetsov, "PSPARLIB users manual: A portable library of parallel sparse iterative solvers", Technical report, University of Minnesota, 1997.
[7] N. Seoane and A. García Loureiro, "Analysis of Parallel Numerical Libraries to Solve the 3D Electron Continuity Equation", *Lecture Notes in Computer Science*, **3036**, 590-593 (2004).
[8] Y. Saad, "Iterative methods for sparse linear systems", PWS Publishing Co. (1996).
[9] Galician Supercomputing Center, http://www.cesga.es
[10] K. Kalna, A. Asenov, K. Elgaid and I. Thayne, *Solid State Electron*, **46**, 631 (2002).
[11] S. A. Vavasis, "QMG 1.1 Reference Manual", Computer Science Department, Cornell University, 1996.
[12] M. Aldegunde, Juan J. Pombo, A. García Loureiro, "Octree–based mesh generation for the simulation of semiconductor devices", XX Conference on Design of Circuits and Integrated Systems (DCIS), 2005.
[13] "Itanium processor family performance tuning guide", Technical paper.
[14] A. Quarteroni and A. Valli, "Domain Decomposition Methods for Partial Differential Equations", Oxford University Press (1999).

Real-time simulation for laser-tissue interaction model

L.F. Romero[a], O. Trelles[a], M.A. Trelles[b]

[a]Dept. Computer Architecture, University of Malaga, Spain

[b]Medical Institute of Vilafortuny, Cambrils, Tarragona, Spain

The extensive use of laser as a medical and surgical tool has lead to a growing interest in modelling the interactions between laser irradiation and human tissues. This modelling has enormous computational needs where the use of parallel computing can provide CPU-power enough to obtain results in real time, and with a proper representation. To this end we have developed a three-layer, parallel architecture in order to simulate the laser-tissue interaction. The first layer is used to obtain the irradiance distribution by Monte Carlo simulation under an optical model; Then, in the second layer, the temperature changes produced by the energy delivered by laser device is obtained by means of a differential equation based model. Finally, the thermal damage is predicted from the spatial and temporal temperature distributions. To achieve high efficiencies and real time results, the complexity of the model requires a complex parallel implementation. In this work, an interface for a hybrid parallel communication model is presented. This interface makes easier the programming of high efficiency hybrid codes, even with a reduced set of processors.

1. Introduction

Laser treatment based on controlled tissue elimination using selective photothermolysis is now well established as the treatment of first choice for various skin lesions and especially for the treatment of pigment disorders and skin tumors. When choosing a proper set of irradiation parameters (wave-length, pulse length, beam size, etc.) for a pulsing laser beam applied to a given target zone, some undesired kinds of tissue can be destroyed by inducing thermal damage in it, while the temperature of the surrounding tissues is kept below the threshold for damage.

However the optimal choice of laser irradiation parameters and guidance of treatment is closely related to the prognosis of results. This is a very complex problem because it is strongly associated with the specific lesion characteristics with an enormous variety in histopathology of skin disorders, the surrounding tissue and its particular structure distribution in the various skin layers, the type of laser device with a diversity of laser irradiation parameters, among others. The many issues involved in this problem made of it a field of actual interest that claims for an in-deep research. In the last, this problem has lead a growing interest in modelling the interaction between laser irradiation and human tissues.

We have address this problem by a new laser/tissue interaction model based on three different layers: (a) First, the irradiance distribution –how light delivered by laser device propagates through such tissue models– is determined by Monte Carlo simulation; next (b) the temperature distribution in the tissue caused by laser energy deposition is estimated by solving the *bioheat* transfer equations; and lastly (c) the thermal damage is predicted from the spatial and temporal temperature distributions, with the aid of so-called damage integral Arrhenius formulation, in which the thermal damage to tissue is described as a temperature dependent rate process. The three layers architecture, presented in Section 2 allows a close reproduction of the effect of laser with a realistic tissue model. Figure 1 shows a schematic representation of the model, on the left, and the most important dependencies among the layers and its parameters, on the right.

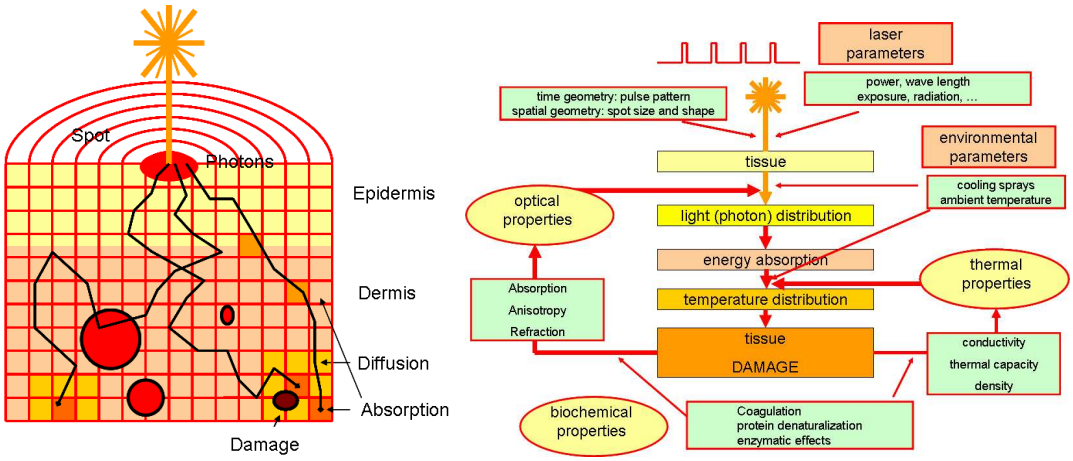


Figure 1. A laser tissue interaction scheme and a dependency diagram of the model.

Parallel computing with low-cost multiprocessor systems have been employed to achieve the real time simulations required for monitoring the laser-tissue interaction. While the Monte Carlo simulation in parallel is trivially solved, the major computational cost and complexity arise from the integration of the equation describing the heat transfer. To obtain a faster computation with good accuracy, a finite differences method in parallel is employed here, using a block distribution strategy for the discretization grid. Finally, Arrhenius formulation takes profit of the grid distribution and no data interchange is necessary in this step. The parallelization of the model is presented in Section 3.

One of the major advantages using non-deterministic techniques in the simulation process lays in the fact that processors synchronization is not strictly necessary to obtain information. Different parallelization techniques and communication paradigms have been combined to optimize response time producing high efficiency and near real time results. In Section 4 an interface to make easier the programming of efficient hybrid codes is presented.

The final integrated application allows to conduct the full simulation in real time (shown in Section 5) after the configuration parameters are settled. A multi-layer description of tissue allows a very close representation of the different tissue irregularities, with a easy definition of the diverse tissue components such as chromospheres, small veins, dermis, epidermis, etc, and even it is possible to incorporate external sources like as cryogen gels or to modify the device parameters.

2. The model

The three layers of the computational scheme of the laser/tissue interaction model are strongly related. As stated above, the photon energy of a pulsing laser in layer 1 increases the temperature in layer 2, and the induced heat modifies the cells in layer 3. As the thermal and optical properties are modified in a damaged tissue, there is a feedback of layer 3 over the first two layers of the proposed model. In order to obtain a realistic model, an time iterative implementation of the three layers is required, in which the time step should be as short as possible. But in practice, the cell degradation by effect of the temperature, and so, the changes of the optical and thermal properties, are governed by a time factor which is much larger than the required precision in the solution of the diffusion equation for typical grid sizes. For this reason, a compromise solution with a relaxed interaction

(with two levels of time steps) among the layers have been considered, in order to minimize the communication overhead of the model and to increase its temporal locality:

```
Algorithm 1:      for i = 1,..., number of global time steps
                     compute photon/energy distribution (layer 1)
                     for j = 1,..., number of diffusion time steps
                          Energy absorption=f(energy distribution,pulsing laser time shape)
                          compute thermal diffusion (layer 2)
                     compute damage (layer 3)
```

2.1. Light distribution

Light distribution is estimated by launching photons governed by the optical properties of tissue (scattering, absorption, etc) [3]. The number of photons must be large enough to obtain statistically validated results. In the travel through the tissue, every photon deploys some of its energy in small cells (*voxels*) resulting from a discretization of the tissue. The optical properties of each *voxel* depends on the tissue layer to which it belongs [1]. The process is represented by this algorithm:

```
Algorithm 2:      for i = 1,..., number of photons
                     launch photon
                     while (photon in grid) and (photon alive)
                          move photon
                          compute scattering
                          if (layer boundary)compute refraction and reflection
                          compute energy deposition
                          if (energy < threshold) photon not alive
```

The Monte Carlo method employed here requires a high computational power for statistically valid simulations, but more accurate results can be obtained in comparison with other methods.

2.2. Thermal difusion

The heat transfer is modelled in this work by the *bioheat* equations with advection (1), and the corresponding contour conditions [5,6]:

$$\frac{\partial T}{\partial t} = \frac{k}{\rho \cdot C}(D \cdot \frac{\partial^2 T}{\partial x^2} + D \cdot \frac{\partial^2 T}{\partial y^2} + \frac{\partial}{\partial z} D \cdot \frac{\partial T}{\partial z} + F_{met} + F_{circ} + A + E_{laser}, \quad (1)$$

where T is the temperature, ρ is the density, C is the specific heat, k is the thermal conductivity, D is the diffusion coefficient; F_{met}, a heat source from the cellular metabolism; F_{circ}, a heat source from the smaller blood vessels; A is the advection (see Figure 2), and finally, E_{laser} is the energy carried by the photons, computed in the previous layer. Due to the typical size of photon sampling and the dimensions of the integration grid for the partial differential equation system, our choice has been pre-computing the energy distribution matrix based on light distribution in the previous step and on the time shape of the pulsing laser.

Equation(1) has been integrated using a Crank–Nicolson finite difference method. It is noteworthy to observe that both spatial and temporal discretization in this step are the same as used in the Monte Carlo simulation. Once the energy distribution is known and the matrix coefficients have

[1] A typical simulation uses $50 \times 50 \times 50$ *voxels*, each one of size $100\mu m \times 100\mu m \times 100\mu m$.

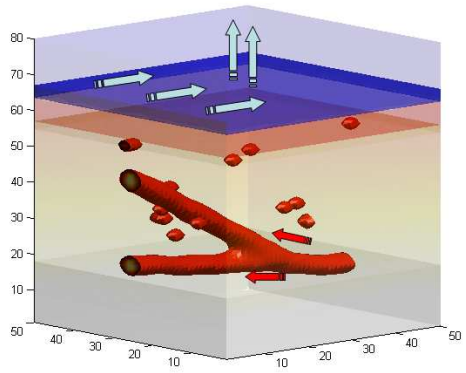


Figure 2. A complex model with an air layer, cryogen gel, epidermis, dermis, subcutaneous tissue, vessels, chromospheres, and several advection terms (represented with arrows).

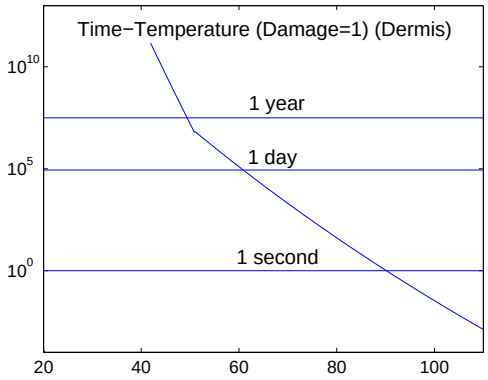


Figure 3. Exposure time required to damage a 100% of dermis cells, as a function of temperature.

been computed, the resulting system of $n = n_x \times n_y \times n_z$ linear equations is solved using the Preconditioned Conjugate Gradient (PCG) method.

2.3. Arrhenius formulation for the tissue damage

The thermal increase modifies the physiological properties of a tissue. Normally, when temperature rises to about 50°C, there is a protein denaturalization which induces the death of cells in short time. Arrhenius formulation [1] is employed here to calculate the accumulative damage ($\Delta\Omega$), which is irreversible when the total induced damage affect to the 100% of cells (Ω=1). Arrhenius equation (2) computes the accumulative damage in a tissue, exposed to a given temperature for a certain time:

$$\Delta\Omega(T,t) = A\int_{t_i}^{t_f} e^{-\frac{E_a}{R\cdot T}}dt \simeq A(t_f - t_i)\cdot e^{-\frac{E_a}{R\cdot T}}, \tag{2}$$

where A is the frequency factor, $t_f - t_i$ is the exposure time, R is the universal gas constant, and E_a is the energy activation barrier. In Figure 3 the time required to an irreversibe damage of dermis tissue is shown for different temperatures. Usually, the damage will be mainly produced in the epidermis (due to its proximity to the laser source), and also in the haemoglobin (because of its high absorption coefficient). The use of cryogen gels (below 0°C) will minimize the damage in the epidermis [2].

3. Parallel implementation

In order to achieve real time realistic simulations, a parallel implementation has been developed, which can be properly scaled in order to get the maximum performance from the available resources. Up to four level of parallelism have been combined in a typical implementation of the model, as described below (Fig. 4):

1) A client-server model, usually using a PC acting as front-end client, sending to a simulator Server the control parameter of the model (such as intensity, pulse shape, environmental conditions, etc.). The front-end client also renders any volume data which have been received from the simulator Server. The experiment presented below have been obtained using a PC with a Pentium 4 as client, and an Altix 3000 as server.

2) The Altix server launches two MPI processes. The first one computes the trajectory of photons (the *photon server*), while the other one solves the bioheat equation (the *grid server*).

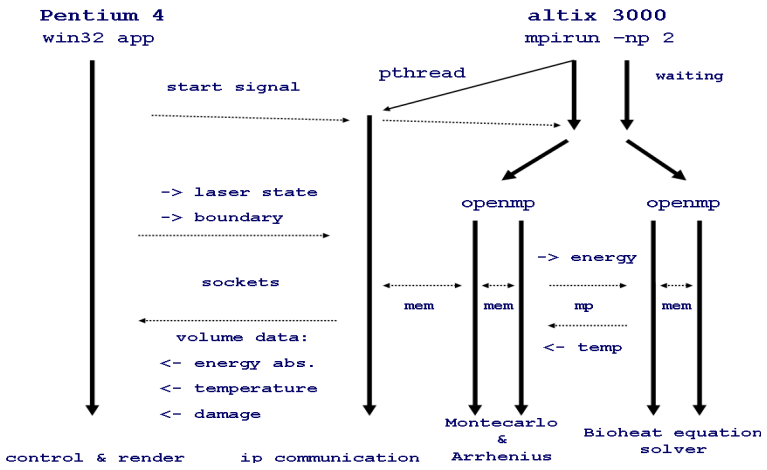


Figure 4. Gantt chart for a typical parallel implementation

3) Both MPI processes have been parallelized using a shared memory model with OpenMP. In the first MPI process, Monte Carlo is parallelised distributing the number of photons among the different processors and collecting results in a master thread which will communicate with the grid server. Photon trajectories are independent allowing an efficient parallelization. In the second one, a block distribution strategy for the discretization grid following the z-direction is used, in such a way that each processor computes the matrix coefficients and the right hand sides in their assigned nodes. This is equivalent to a block distribution of the system matrix if a natural ordering is used. The resulting system is solved by means of the PCG method, in which each vectorial operation is solved in parallel by a block distribution of the vector components among processors.

4) In addition, one of the MPI processes (usually the *photon server*) launches a Posix thread, which will perform an asynchronous communication with the PC client.

3.1. Communication pattern

Note that the communication between the processes and threads may be synchronous or asynchronous, depending of the requirements in each level. It has to be taken into account that, like reality, this is an stochastic model, in which the computational rigourousness in the solution of the problem can be partially sacrificed without a penalty in the realism of the results. So, the PC client and the server machine only requires asynchronous communication: the posix thread will continuously send the data as they are found in the memory, in any moment, and it will forward the control commands received from the PC to a memory location in one of the MPI processes. The volume data for visualization is sent by the server using just one byte per voxel in a compressed package. Decompression and rendering of the volume data is performed by an event driven application using the fast VTK visualization library [7]. The communication rate only depends on the LAN connection and the frequency of this asynchronous communication is large enough to avoid a visual mismatch.

The *photon server* and the *grid server* also communicates each other by using an asynchronous message passing model (MPI_Isend, MPI_Irecv) at a frequency of about 10Hz. This frequency is large enough to eliminate numerical instability and to provide realistic simulations. Note that the protein denaturalization, which modifies the optical parameters for a temperature increase, usually occurs at a rate of tens of seconds. On the other hand, communications in the *grid server*, required for the solution of equation (1), which occurs through the memory, must be carefully synchronized.

3.2. Load balancing

An efficient parallelization strategy for the PCG solver has been used in this work, which will ensure a minimization of the communication cost and a good load balancing inside the *grid server* [4]. As the integration of equation (1) establishes the temporal reference for the system, the computational load of the *photon server* can be easily adjusted (with a minimum threshold, statistically stablished), depending on the employed number of threads.

4. A communication model for hybrid systems

In this section, an interface to make easier the hybrid programming of such a complex parallel model is presented. The proposed idea is to make an abstraction of both the data interchange and the synchronization requirements, by solving the underlying problem in any parallel programming paradigm: the read-after-write (RAW) and write-after-read (WAR) hazards for any block of data being shared by two or more processors. Note that all parallel programming environment existing in the literature shares the information by using one of the following strategies: a) storing information in a shared memory location; b) copying remote information in a local storage: *get*; c) copying local information in a remote storage: *put*; and finally, by using a *send-receive* symmetric communicator.

The interface proposed here is based in the substitution of the mentioned operation (*put, get, send, recv*) and of any required syncronization (flags, semaphores, locks, etc.), by the use of only four routines, which has to be carefully inserted after and before the use of shared data. These routines may use any of the traditional strategies inside, depending of a the requirements, as explained below:

- *pre-write*: Routine used to evaluate a WAR hazard just before any modification of local data which is being shared. This call is only required in a shared memory environment, or when a *get* communicator is employed. Usually, it consist in evaluate if a certain *flag* variable,described below, is zero (WAR check).

- *post-write*: This routine is used after modifying the data, and includes three stages. Firstly, WAR hazard is evaluated, only if a *put* communicator is employed. It evaluates the availability of a remote location. Second, the data is delivered if required, by using *send* or *put*. And third, a *flag* variable is lifted, as the initial step of a RAW hazard detection, if a symmetric communicator is not being used (RAW release).

- *pre-read* : Routine used to receive data before any computation using it. It also includes three stages. Firstly, the *flag* variable is tested (if a symmetric communicator is not being used), as the final step of a RAW hazard detection. Second the data is received, by using *recv* or *get*. And third, the *flag* variable is done zero, as the liberation phase of a WAR hazard detection. The third stage is only required when a *get* communicator is employed. (RAW check).

- *post-read*: Routine used to liberate a WAR hazard just after using data produced in another processor. The *flag* variable is done zero. This call is only required in a shared memory environment, or when a *put* communicator is employed (WAR release).

In Table 1, the four operations of the proposed interface are summarized. Note that these routines should be applied to large blocks of data, rather than to individual variables (in the same way that messages should be unified, if possible, in a message passing model). For example, a *post-write* call should be invoked just after the last local modification, before an access from a remote processor. In general, the routines presented here should be placed immediately after or before the read or write operations, to ensure a minimization of the waits, and to overlap communications and computations.

The synchronization flags described here can be reused by different communication operations with some care. For example, in a global reduction, all flags involved can be replaced by a single counter. Also, if a same operation is performed inside a loop, an even-odd pair of flags should be alternatively used, to prevent from hazards affecting to the flag variable. Finally, when any of the paired routines (either a *post-read/pre-write* for WAR hazards, or a *post-write/pre-read*, for a RAW hazard) are dynamically separated by another pair, involving, at least, at the same PEs, then the synchronization operations over the flags in the first pair can be eliminated, because the hazard has been solved by the second one. So, many of the WAR hazard detections can be eliminated.

By considering all this items together, the resulting code should minimize the communication cost, and it will significatively reduce the waits in the synchronization points due to load unbalance.

Communicator	pre–write	WR	post-write	pre-read	RE	post–read
shared	test flag==0;		flag=1;	test flag==1;		flag=0;
symm MP			send;	recv;		
put			test flag==0; put; flag=1;	eval if 1		flag=0;
get	test flag==0;		flag=1;	test flag==1; get; flag=0;		

Table 1

PEs	task distribution	Ex. time(grid server)	Launched photons	speedup MC	speedup ED
1	1 (photon server)	-	13050	1	-
2	1 (photon server), 1 (grid server)	4.88 sec.	63650	-	1
4	1 (photon server), 3 (grid server)	0.98 sec.	12950	0.99	4.98
8	2 (photon server), 6 (grid server)	0.50 sec.	26550	2.02	9.76

Table 2

5. Results

In this section, we present some results, which has been obtained using an Altix 3000 system as the simulator server. Several experiments has been performed, using a typical skin model with 3 vessels, several chromospheres, and a discretization grid of 125000 voxels (see Figure 5). In all cases, times are shown for a 1 second simulation (10 global times steps, 10×20 diffusion time steps). The first experiment (using 1 PE as a photon server), has been used to determine how many photons can be launched by 1 PE in one CPU second. In the second and third experiments, one of the Itanium processor has been used as a photon server, while 1 and 3 (respectively) CPUs has been used for the solution of the bioheat equation (the CPU usage for the Arrhenius problem is irrelevant). In the third experiment, the parallel efficiency obtained (1.66) is very high, which is not only due to the cache effect, but also to a total absence of waits in the synchronization points. This is thanks to the use of the technique described in the previous section, which has allowed to eliminate all global barriers, and any unnecessary test for hazards. Real time results have been obtained in this case (note that many of the model properties have been previously adjusted to reach a real time simulation for this configuration of the system). The last experiment indicates that the scalability of the problem is very good for a larger number of processors, thus allowing a more complex modellization of the problem.

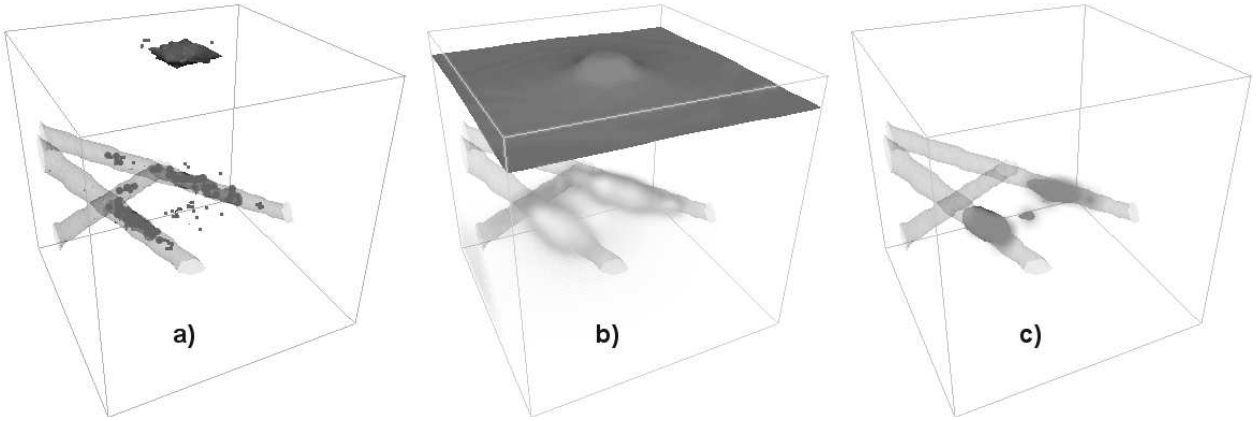


Figure 5. Simulation snapshots: a) photon energy deposition, b) tissue temperature, and c) induced damage for a sample model with three vessels. Figure b) includes a 0°C isosurface in the epidermis, and an opacity render (centered in 90°C) which is mainly located around the vessels.

6. Conclusions

In this work, an integrated model for the laser-tissue interaction has been presented. By means of the three-layer architecture of the model, an accurate representation of the tissue response to the laser stimulus can be achieved because of the extended set of parameters considered. Also, the three-layer architecture of the model incorporates a complex multi-level parallelism. Several communication paradigms and synchronization techniques have been included, because of the requirements of each level. To make easy the programming of the model, a new parallel programming interface for communications and synchronizations is proposed, which is independent of the parallelization paradigm.

With this interface, a robust, efficient and accurate parallel model has been obtained, which can achieve real time simulations of a complex model, even using a small parallel architecture. These results represent an step forward in assist greatly in diagnosis and treatment, and subsequently objectively assess the device parameters at various stages of treatment.

7. acknowledgement

The authors wish to thank Victor Espigares for his help in the integration of the model.

References

[1] J.K.Barton, A.Rollins, S.Yazdanfar, T.J.Pfefer, V. Westphal, J. A.Izatt, "Photothermal coagulation of blood vessels", *Physics in Medicine and Biology*, vol.46, pp.1665-1678, 2001.

[2] T.J. Pfefer, D.J. Smithies, T.E. Milner, M.J. Van Gemert, J.S. Nelson, A.J. Welch, "Bioheat transfer analysis of cryogen spray cooling during laser treatment of port wine stains", *Lasers in Surgery and Medicine*, vol.26, pp.145-157, 2000.

[3] A. Roggan, G. Müller, *Dosimetry and Computer Based Irradiation Planning for Laser-Induced Interstitial Thermotherapy*, SPIE Institute Series IS 13, Müller-Rogan Eds., 1995.

[4] L.F. Romero, E.M. Ortigosa, J.I. Ramos, "Parallel scheduling of the PCG method for banded matrices rising from FDM/FEM", *Journal of Parallel and Distributed Computing*,vol.63, pp.1243-1256, 2003.

[5] M.J. Van Gemert, A.J. Welch, J.W. Pickering, *Modelling Laser Treatment of Port Wine Stains*, O.T.Tan Eds., Amsterdam, 1992.

[6] M.J. Van Gemert, *Optical-Thermal Response of Laser-Irradiated Tissue*, Plenum, London, 1995.

[7] L. Avila *et al,* The VTK User's Guide*, Kitware Inc., 2004.*

Computer Simulation of the Acoustic Impedance of Modern Orchestral Horns

A. Benoit[a], J.P. Chick[b]

[a]School of Informatics, The University of Edinburgh, James Clerk Maxwell Building, The King's Buildings, Mayfield Road, Edinburgh EH9 3JZ, UK. abenoit1@inf.ed.ac.uk

[b]School of Engineering and Electronics, The University of Edinburgh, Sanderson Building, The King's Buildings, Mayfield Road, Edinburgh, EH9 3JL, UK. john.chick@ed.ac.uk

This paper outlines the development of a computational tool for modelling the acoustic impedance of modern orchestral (French) horns. The acoustic behaviour of an orchestral horn differs significantly from other brasswind instruments because of the presence of the players right hand in the bell of the instrument. We propose to model the acoustic wave propagation in the complex horn/hand geometry using finite-difference techniques, and we explain how the use of a high-level parallel programming library allows us to parallelise the resulting, processor intensive, computation.

Keywords: horn, brasswind, acoustic impedance, finite-difference, high-level parallel library, parallel application.

1. Introduction

The sound propagation in an enclosed space depends on a variety of closely coupled and nonlinear parameters and is thus often a complex subject. This is typically the case for musical instruments. The advent of easily accessible computational tools includes the development of acoustic models, constructed in order to make the problems of design and analysis of instrument performance more tractable. These models invariably involve abstractions and simplifications depending on the basis of the particular model. The abstraction and simplification of some of the features in the model can however result in a significant difference between modelled output and observed behaviour for a particular instrument design.

We focus in this paper on the study of brasswind instruments, more specifically orchestral horns. The complex behaviour of these instruments offers an interesting challenge both for acousticians and computer scientists.

A number of studies have been performed to simulate the acoustic input impedance of such instruments resulting from a given bore profile. The input impedance is normally regarded as the principal characteristic for determining the playing attributes of a brass instrument. However, such studies have so far been largely limited to trumpets and trombones since the bore profile is independent of the player for these instruments. The french horn offers an additional challenge for computer modelers due to the presence of the player's hand in the bell. This has a significant impact on the acoustic impedance and should not be ignored in modelling work. Having the player's hand partially inserted into the bell of the instrument has two significant consequences for the modeller: first, the problem may no longer be regarded as axisymmetric, and a three-dimensional model may be needed to provide an adequate solution; secondly, the playing characteristics of the instrument are now more closely coupled to the player through his/her hand position. This latter problem is outside the scope of this paper but is part of an ongoing project at the University of Edinburgh.

We thus propose to model the impedance of a horn taking into account the presence of a hand in the bell. The acoustic wave propagation in the complex horn/hand geometry is modelled using

conventional finite-difference techniques, and the computations are parallelised in order to make the highly computer intensive simulations feasible for the instrument designer, i.e. there is an acceptably short run time. Although many models have been proposed for brasswind instruments, to the best of our knowledge none has been developed to take into account the effects of the hand and reflect the complexity of the instrument/player coupling in this context.

Structure of the paper

The next section provides further motivation for this work, showing, with the aid of experimental measurements, the lack of agreement between the standard transmission line models and measured data for the case of the french horn. We present the one dimensional transmission line theory and show that the results may not be satisfactory for modelling the impedance of orchestral horns. Section 3 proposes a model of the wave propagation in the complex geometry resulting from the hand in the bell. Since this model is highly processor intensive, we propose a suitable regime for parallelising these computations. To facilitate the process, tools and techniques from high level parallel programming are used. Finally, we conclude and detail future work in Section 4.

2. Acoustic impedance of brass instruments and the horn

Our work is largely motivated by our experiences of standard models of acoustic impedance of brass instruments not providing satisfactory results for the case of the horn, typically because of the presence of the player's hand in the bell of the instrument, and the rapidly flaring bell section in general. We briefly outline in the next section the classical one dimensional transmission line (1-D TL) theory and how we model it using MATLAB [3]. Section 2.2 compares the model analysis with experimentally measured data, and illustrates some of the deficiencies of using this approach.

2.1. 1-D Transmission line theory

A number of computational models of wind instruments have been developed from one dimensional transmission line (1-D TL) theory as discussed by Benade and Jansson [4], Keefe [8], and van Walstijn [14]. These models provide an excellent first estimate of the acoustic impedance of the instrument but their accuracy is somewhat limited in the region of rapidly flaring bell sections. Noreland [10,11] presents a composite model using conventional 1-D TL analysis for the narrow, slowly tapering sections, and couples this to an axisymmetric finite-difference model of the rapidly flaring bell section, with results to good effect.

The 1-D transmission line model described here is based on a piecewise solution to the linearised lossless wave equation as described in [14]:

$$\nabla^2 p = \frac{1}{c^2}\frac{\partial^2 p}{\partial t^2} \tag{1}$$

where p is the acoustic pressure, c is the wave velocity, and ∇^2 is the Laplacian operator for a cartesian coordinate system. For plane waves propagating in one dimension, the wave equation reduces to:

$$\frac{\partial^2 p}{\partial x^2} = \frac{1}{c^2}\frac{\partial^2 p}{\partial t^2} \tag{2}$$

Corrections can be made to Equation (1) and Equation (2) to account for visco-thermal losses at the boundary surfaces of the horn.

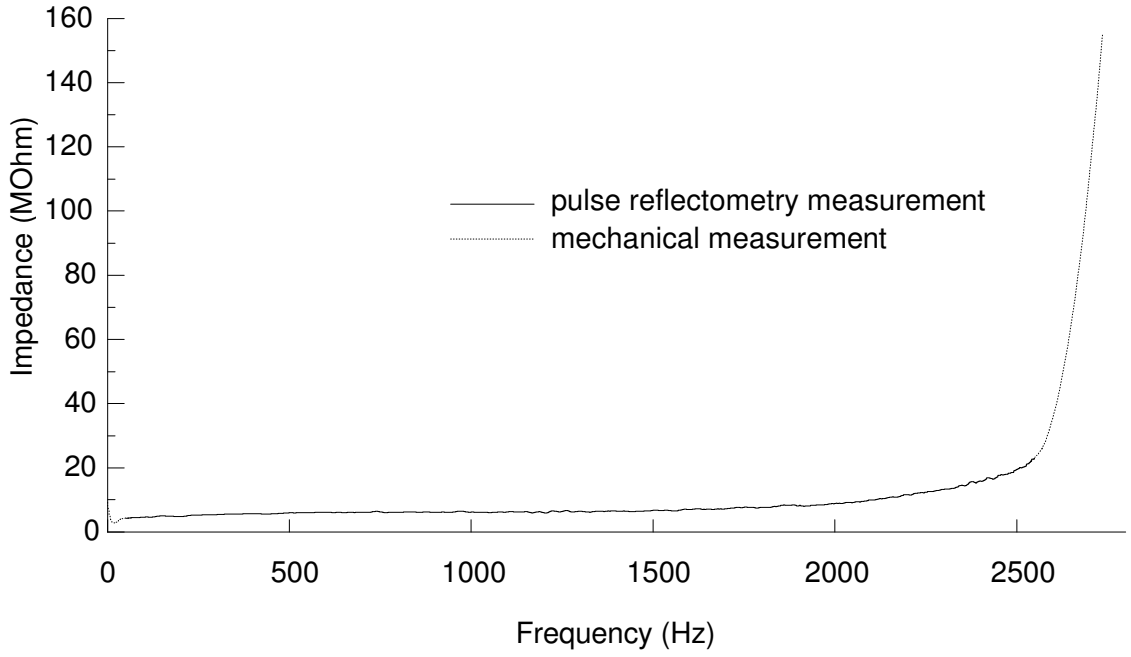


Figure 1. Measurement of the internal bore profile of the open B♭ tubing of a Paxman model 40 B♭/f-alto horn.

There are no analytic solutions for the general case of horns of varying cross section, but a lumped parameter or piecewise solution, in which the horn is modelled as a series of short cylindrical or conical sections, yields good results for horns with only moderately flaring bell profiles. The classical method of piecewise modelling of wind instruments is described by Plitnik and Strong [13]. Solutions are usually expressed in terms of acoustic impedance, $Z(\omega)$, defined as the ratio of the acoustic pressure $p(\omega)$ and the acoustic volume velocity $U(\omega)$ measured at the input plane of the instrument for a sinusoidal input signal of angular frequency ω. These 1-D transmission line models become less effective in the region of rapidly flaring bell sections principally due to the excitation of higher order modes.

The MATLAB model used here reads in a text file containing information about the bore profile: axial distances from the mouthpiece and corresponding bore radii. The model outputs frequency and acoustic impedance.

A comparative study between output from the transmission line model and measured data for a modern orchestral horn highlights this weakness. The bore profile of an instrument (including mouthpiece) was measured using a combination of acoustic reflectometry and traditional mechanical measurements. Pulse reflectometry measurements were obtained using apparatus developed at the University of Edinburgh by Kemp [9]. This technique has proved useful in obtaining accurate bore profile data of brass instruments up to (but so far not including) the final rapidly flaring region of the bell. The bell and mouthpiece were measured using conventional mechanical techniques. The instrument studied here is the open B♭ tubing of a Paxman model 40, B♭/f-alto horn. The bore profile is shown in Figure 1.

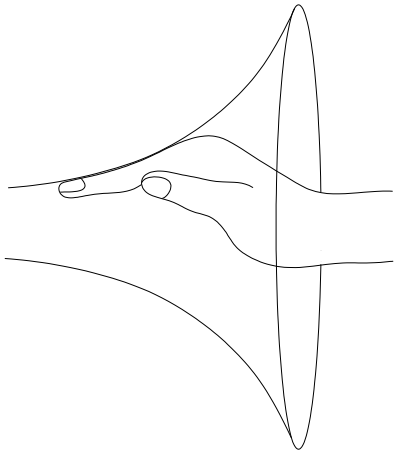


Figure 2. Typical position of a player's hand in the bell of a French horn.

2.2. Specific issues of horn technique

Unlike other members of the brasswind family of musical instruments, the horn is played with the player's hand partially inserted into the bell of the instrument as shown in Figure 2. This is a legacy of historical practise developed prior to the invention of the valve. Where, by varying the level of obstruction of the bell throat by the hand, the pitch of an "open" note may be manipulated to provide notes that would not otherwise be available on a fixed length of tubing. By modifying the termination impedance of the horn the presence of the hand in the bell also extends the range of resonant modes in the upper register of the instrument, typically from B♭4 and above. Without the presence of the hand in the bell, the high register becomes difficult or impossible to play. Thus, the player's hand forms an integral part of the instrument, and it is difficult to assess the quality of a particular instrument without accounting for this.

Figure 3 illustrates measured impedance curves taken from the open tubing of a B♭ horn with the hand positions of two different players, and without the hand in the bell. The playing response and intonation of the instrument is very strongly dependant on the frequencies and magnitudes of the local impedance maxima.

This first experiment clearly shows the impact of the hand position in the bell and its effect on the acoustic impedance. Further investigation illustrates how the 1-D TL model may not be acceptable when applied to a study of the acoustic impedance of the horn. The curves in Figure 4 show the difference between the 1-D TL model prediction of Section 2.1 and the measurements for the same instrument, and we can see that there is a clear difference between those curves, particularly in the high register.

3. Numerical modelling of the bell/hand geometry

Section 2.2 clearly illustrated the need for any model whose function is to calculate the acoustic impedance of the horn to take into account the player's hand, its shape and position in the bell. Thus, the model needs to be more sophisticated than the standard 1-D TL model. In this section we describe some features of how to model the bell/hand geometry, and present a user friendly design tool for instrument makers, based on this model.

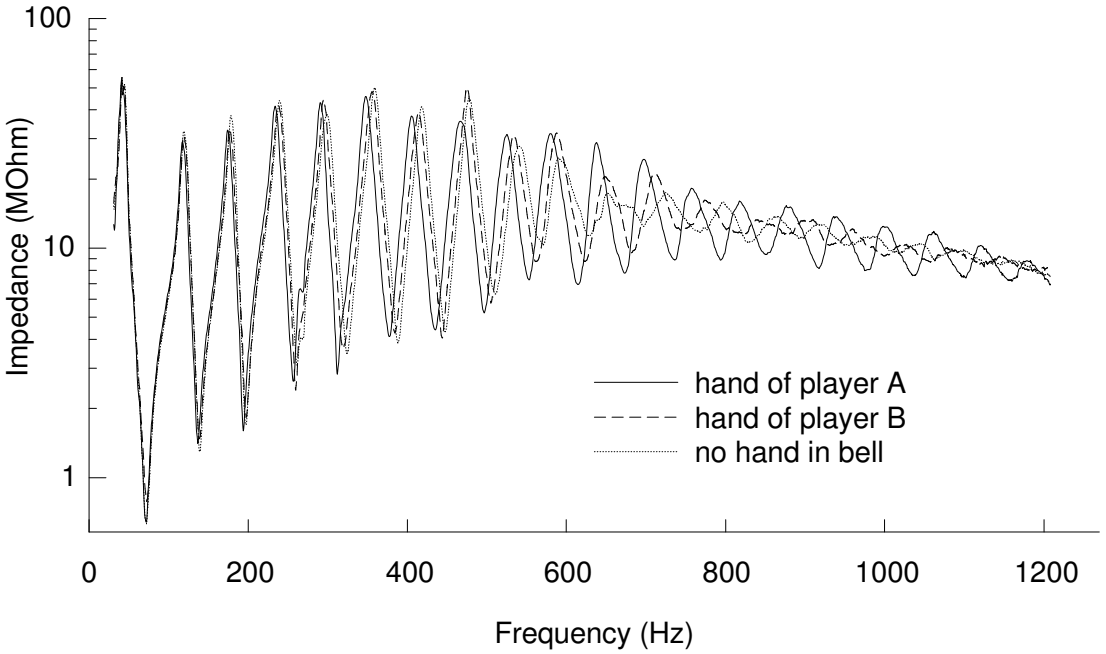


Figure 3. Impedance curves taken from the open B♭ tubing of an Alexander model 103 F/B♭ horn for different player's hand positions.

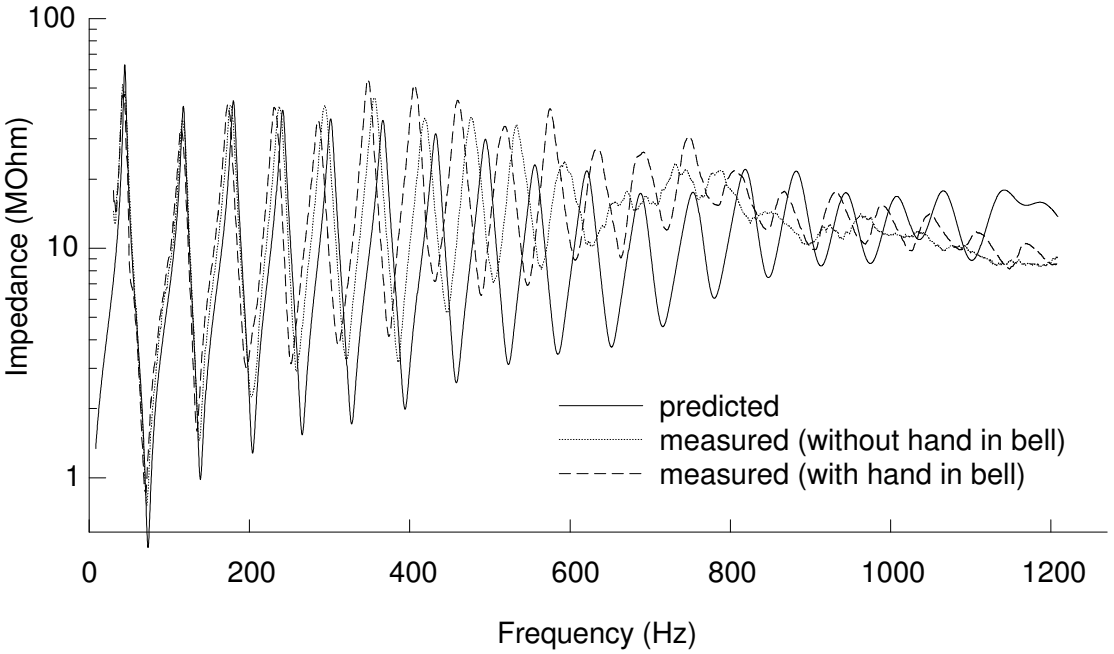


Figure 4. Impedance curves taken from the open B♭ tubing of a Paxman 40 B♭/f-alto horn, with and without hand, and 1-D TL model curve.

3.1. Basics of the model and tool prototype

To the authors' knowledge no computational study has been conducted which includes the effects of the presence of the player's hand in the bell of an orchestral horn. The solution proposed by Noreland [10,11] for rapidly flaring axisymmetric bell sections provides an excellent starting point and may be adapted for this purpose by modelling the bell/hand geometry using a 3-D finite-difference scheme.

Modelling wave propagation in the complex geometry produced by the hand in the bell is highly processor intensive and can result in long run times. For this reason, the modelled domain is decomposed into a number of subdomains, for each of which the wave equation may be solved using conventional finite-difference techniques.

As part of an ongoing project to develop a user friendly design tool for instrument makers, we propose to parallelise the problem solution as detailed in the next section. This parallelised routine is at the heart of a linked suite of computer programmes currently under development. The prototype of our tool is displayed in Figure 5.

The main component of the suite is *ProCAIB* (**Pro**gram for **C**alculating the **A**coustic **I**mpedance of **B**rasswinds). Input to *ProCAIB* is generated by running the preprocessor application, *ProGIG* (**Pro**gram for **G**enerating the **I**nstrument **G**eometry files).

ProGIG is parameterised by a series of easily modifiable generic design templates that the instrument designer can manipulate to his or her needs. In the case of the horn, the hand position is also taken into account and profiled through the preprocessor.

The output of *ProCAIB* consists of the acoustic impedances corresponding to the given bore profiles/hand geometry. A final component in the suite is the post processor *ProANAB* (**Pro**gram for **ANA**lysing **B**rasswinds data). *ProANAB* assists in the interpretation of modelled output, providing feedback to the instrument designer. Typically, instrument design is an iterative process and output from *ProANAB* can be used directly to modify the input to *ProGIG*, taking into account the results from earlier computations to refine the profiling of the instrument.

The main computational part of the problem is in the *ProCAIB* component, for which we propose a parallel algorithm to deal with the numerous parameters.

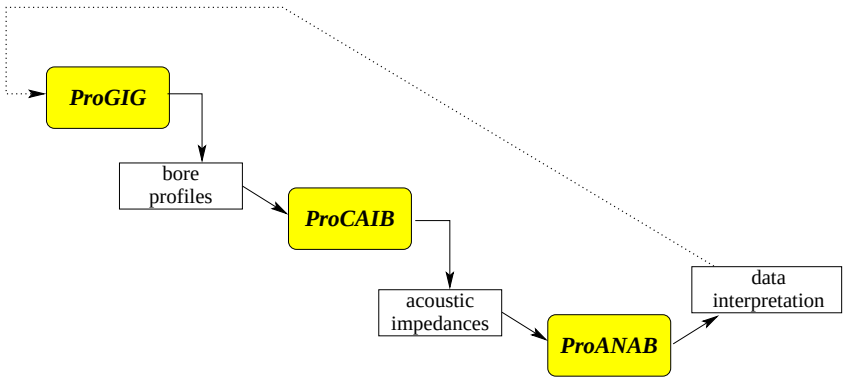


Figure 5. Prototype of a tool for instrument designers.

3.2. Parallelising the model

We propose the development of an ad-hoc solver for the solution to the wave equation, using a high-level parallel programming approach integrating MATLAB [3] numerical algorithms. This allows us to separate the application-specific design from the parallelisation problem. Our goal is to parallelise the *ProCAIB* component.

Several high-level approaches to parallel programming have been documented in the research literature [7,12], based on the fact that many parallel algorithms are following a number of generic patterns (or *skeletons*) of computation and interaction. The skeletal programmers abstract such patterns, and provide them to the application programmer as a library, allowing the programmer to experiment easily with a variety of parallel structures for a given application, without needing to attend to the underlying implementation of the parallel computations and interactions.

Calls to MATLAB functions can be made from a C program through the use of the MATLAB Engine [2]. We therefore propose to perform calls to MATLAB functions from the C/MPI-based skeleton library *eSkel* [5]. In this way, we can have direct control over the interactions between the different parts of the model, using either implicit or explicit interactions [6], and we can refine the parallel scheme of the program.

We propose to parallelise the computation for a given bore profile/hand geometry, since we have identified these to follow a *pipeline* pattern. The model can be decomposed into several sub-domains, and several successive computations must be performed on each of these sub-domains. The pipeline skeleton allows us to associate a state to each pipeline stage, and this state is evolving while processing the successive sub-domains, taking into account the correlations between these computations.

For automated design optimisation work we can add an additional level of parallelism in which several bore profiles may be analysed in order to determine the best design solutions for an instrument within a given design space. In this case, the parallelisation is straightforward and easy to integrate with *eSkel* using the *farm* skeleton to distribute the independent computations on several processors. The MATLAB Distributed Computing Toolbox [1] could be used for the same purpose, since it allows several executions of a MATLAB program to be run in parallel on a cluster of computers. However, our choice is motivated by several arguments:

- the implementation is straightforward because of the facilities offered by high-level parallel programming libraries;
- we can integrate the inner parallelisation of the computation for one single bore profile;
- we have a better control on the allocation of processes onto the available processors and on the parallel behaviour of the program;
- all the parallel software used is a free-software.

4. Conclusions and future work

In this paper, an overview is presented of the complex problem of modelling the bell/hand geometry of an orchestral horn. Some experimental measurements have been done to show the impact of the presence of the hand in the bell on the acoustics of the horn. We have also shown that the classical one dimensional transmission line model may not be satisfactory for the study of the horn, and that a more complex model is needed for this instrument.

Such models can be developed using finite-difference techniques, and we propose to parallelise the simulation algorithm using a high-level parallel library. Indeed, the model is complex and resolution is computationally highly intensive and would be cumbersome if performed in a sequential manner.

Initial tests to perform calls to MATLAB functions in parallel through the use of a parallel library have shown promising results in the case of the 1D-TL models, with the help of the MATLAB Engine. This work is ongoing, and further work remains to be done before we can fully assess the performance of the parallel algorithm based on the simulation using the finite-difference techniques.

To conclude, we have shown that a new model is needed to assist in the development of orchestral horns, and we have outlined the design of such a model. Having tested the integration of MATLAB models with the parallel library, we believe that our methodology is robust and will result in a useful and reliable design tool.

References

[1] MATLAB Distributed Computing Toolbox. `http://www.mathworks.com/products/distribtb`.

[2] MATLAB Engine. `http://www.mathworks.com/access/helpdesk/help/techdoc/matlab_external`.

[3] MATLAB. `http://www.mathworks.com/products/matlab`.

[4] A.H. Benade and E.V. Jansson. On plane and spherical waves in horns with non uniform flare. I. Theory of radiation, resonance frequencies and mode conversion. *Acustica*, 31(2):79–98, 1974.

[5] A. Benoit and M. Cole. The Edinburgh Skeleton Library eSkel, 2005. `http://homepages.inf.ed.ac.uk/abenoit1/eSkel`.

[6] A. Benoit and M. Cole. Two fundamental concepts in skeletal parallel programming. In P. Sloot V. Sunderam, D. van Albada and J. Dongarra, editors, *The International Conference on Computational Science (ICCS 20 05) , Part II*, LNCS 3515, pages 764–771. Springer Verlag, 2005.

[7] M. Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30(3):389–406, 2004.

[8] D.H. Keefe. Acoustical wave propagation in cylindrical ducts: Transmission line parameter approximations for isothermal and nonisothermal boundary conditions. *JASA*, 75(1):58–62, 1984.

[9] J.A. Kemp. *Theoretical and experimental study of wave propagation in brass musical instruments.* PhD thesis, Acoustics and Fluid Dynamics Group, University of Edinburgh, Scotland, 2002.

[10] D. Noreland. A Numerical Method for Acoustic Waves in Horns. *Acta Acustica*, 88(4):576–586, 2002.

[11] D. Noreland. *Numerical techniques for acoustic modelling and design of brass wind instruments.* PhD thesis, Dept. of Information Technology, Uppsala University, Uppsala, Sweden, 2003.

[12] S. Pelagatti. *Structured Development of Parallel Programs.* Taylor & Francis, ISBN 0-7484-0759-6, London, 1998.

[13] G.R. Plitnik and W.J. Strong. Numerical method for calculating input impedances of the oboe. *Journal of the Acoustical Society of America*, 65(3):816–825, 1979.

[14] M. van Walstijn. *Discrete-Time Modelling of Brass and Woodwind Instruments with Application to Musical Sound Synthesis.* PhD thesis, Faculty of Music, University of Edinburgh, Scotland, 2002.

Parallelization of the C++ Navier-Stokes Solver DROPS with OpenMP

Christian Terboven[a], Alexander Spiegel[a], Dieter an Mey[a], Sven Gross[b], Volker Reichelt[b]

[a]Center for Computing and Communication, RWTH Aachen University, 52074 Aachen, Germany

[b]Institut für Geometrie und Praktische Mathematik, RWTH Aachen University, 52056 Aachen, Germany

1. Introduction

The Navier-Stokes solver DROPS [1] is developed at the IGPM (Institut für Geometrie und Praktische Mathematik) at the RWTH Aachen University, as part of an interdisciplinary project (SFB 540: Model-based Experimental Analysis of Kinetic Phenomena in Fluid Multi-phase Reactive Systems [2]) where complicated flow phenomena are investigated. The object-oriented programming paradigm offers a high flexibility and elegance of the program code, facilitating development and investigation of numerical algorithms. Template programming techniques and the C++ Standard Template Library (STL) are heavily used.

In cooperation with the Center for Computing and Communication of the RWTH Aachen University a detailed runtime analysis of the code has been carried out and the computationally dominant program parts have been tuned and parallelized with OpenMP.

The UltraSPARC IV- and Opteron-based Sun Fire SMP-Clusters have been the prime target platforms, but other architectures have been investigated, too. It turned out that the sophisticated usage of template programming in combination with OpenMP is quite demanding for many C++ compilers. We observed a high variation in performance and many compiler failures.

In chapter 2 the DROPS package is described briefly. In chapter 3 we take a look at the performance of the original and the tuned serial code versions. In chapter 4 we describe the OpenMP parallelization and its performance. Chapter 5 contains a summary of our findings.

2. The DROPS multi-phase Navier-Stokes solver

The aim of the ongoing development of the DROPS software package is to build an efficient software tool for the numerical simulation of three-dimensional incompressible multi-phase flows. More specifically, we want to support the modeling of complex physical phenomena like the behavior of the phase interface of liquid drops, mass transfer between drops and a surrounding fluid, or the coupling of fluid dynamics with heat transport in a laminar falling film by numerical simulation. Although quite a few packages in the field of CFD already exist, a black-box solver for such complicated flow problems is not yet available.

From the scientific computing point of view it is of interest to develop a code that combines the efficiency and robustness of modern numerical techniques, such as adaptive grids and iterative solvers, with the flexibility required for the modeling of complex physical phenomena.

For the simulation of two-phase flows we implemented a levelset technique for capturing the phase interface. The advantage of this method is that it mainly adds a scalar PDE to the Navier-Stokes system and therefore fits nicely into the CFD framework. But still, the coupling of the phase interface with the Navier-Stokes equations adds one layer of complexity.

Several layers of nesting in the solvers induced by the structure of the mathematical models require fast inner-most solvers as well as fast discretization methods since many linear systems have

to be regenerated in each time step. Apart from the numerical building blocks, software engineering aspects such as the choice of suitable data structures, in order to decouple the grid generation and finite element discretization (using a grid based data handling) as much as possible from the iterative solution methods (which use a sparse matrix format), are of main importance for performance reasons.

3. Portability and Performance of the Serial Program Version

3.1. Platforms

The main development platform of the IGPM is a standard PC running Linux using the popular GNU C++ compiler [3]. Because this compiler does not support OpenMP, we had to look for adequate C++ compilers supporting OpenMP on our target platforms. Table 1 lists compilers and machines which we considered for our tuning and parallelization efforts. It also introduces abbreviations for each hardware-compiler-combination, which will be referred to as ”platforms” in the remainder of the paper.

The programming techniques employed in the DROPS package (Templates, STL) caused quite some portability problems due to lacking standard conformance of the compilers, therefore the code had to be patched for most compilers. Unfortunately not all of the available OpenMP-aware compilers were able to successfully compile the final OpenMP code version.

From the early experiences gathered by benchmarking the original serial program and because of the good availability of the corresponding hardware, we concentrated on the OPT+icc and USIV+guide platforms for the development of the OpenMP version and recently on OPT+ss10 and USIV+ss10.

3.2. Runtime profile

The runtime analysis (USIV+guide) shows that assembling the stiffness matrices (SETUP) costs about 52% of the total runtime, whereas the PCG-method including the sparse-matrix-vector-multiplication costs about 21% and the GMRES-method about 23%. Together with the utility routine LINCOMB these parts of the code account for 99% of the total runtime. All these parts have been considered for tuning and for parallelization with OpenMP.

It must be pointed out that the runtime profile heavily depends on the number of mesh refinements and on the current timesteps. In the beginning of a program run the PCG-algorithm and the matrix-vector-multiplication take about 65% of the runtime, but because the number of iterations for the solution of the linear equation systems shrinks over time, the assembly of the stiffness matrices is getting more and more dominant. Therefore we restarted the program after 100 time steps and let it run for 10 time steps with 2 grid refinements for our comparisons.

3.3. Data Structures

In the DROPS package the Finite Element Method is implemented. This includes repeatedly setting up the stiffness matrices and then solving linear equation systems with PCG- and GMRES-methods.

Since the matrices arising from the discretization are sparse, an appropriate matrix storage format, the CRS (compressed row storage) format is used, in which only nonzero entries are stored. It contains an array *val* - which will be referred to later - for the values of the nonzero entries and two auxiliary integer arrays that define the position of the entries within the matrix. The data structure is mainly a wrapper class around a `valarray<double>` object, a container of the C++ Standard Template Library (STL).

Unfortunately, the nice computational and storage properties of the CRS format are not for free. A

machine	platform	compiler	runtime[s] original	runtime[s] tuned	OpenMP support
Standard PC: 2x Intel Xeon 2.66 GHz Hyper-Threading	XEON+gcc333	GNU C++ V3.3.3[3]	3694.9	1844.3	no
	XEON+gcc343	GNU C++ V3.4.3	2283.3	1780.7	no
	XEON+icc81	Intel C++ V8.1[4]	2643.3	1722.9	yes
	XEON+pgi60	PGI C++ V6.0-1	8680.1	5080.2	yes
Sun Fire V40z: 4x AMD Opteron 2.2 GHz	OPT+gcc333	GNU C++ V3.3.3	2923.3	1580.3	no
	OPT+gcc333X	GNU C++ V3.3.3 64bit	2167.8	1519.5	no
	OPT+icc81	Intel C++ V8.1	2404.0	1767.1	yes
	OPT+icc81X	Intel C++ V8.1 64bit	2183.4	1394.0	fails
	OPT+pgi60	PGI C++ V6.0-1[5]	6741.7	5372.9	yes
	OPT+pgi60X	PGI C++ V6.0-1 64bit	4755.1	3688.4	yes
	OPT+path20	PathScale EKOpath 2.0[6]	2819.3	1673.1	fails
	OPT+path20X	PathScale EKOpath 2.0, 64bit	2634.5	1512.3	fails
	OPT+ss10	Sun Studio C++ V10 update1[7]	3613.1	2431.0	yes
Sun Fire E2900: 12x UltraSPARC IV 1.2 GHz (dual core)	USIV+gcc331	GNU C++ V3.3.1	9782.4	7845.4	no
	USIV+ss10	Sun Studio C++ V10 update1	7673.7	4958.7	yes
	USIV+guide	Intel-KSL Guidec++ V4.0[8]	7551.0	5335.0	yes
IBM p690: 16x Power4 1.7 GHz (dual core)	POW4+guide	Intel-KSL Guidec++ V4.0	5535.1	2790.3	yes
	POW4+gcc343	GNU C++ V3.4.3	3604.0	2157.8	no
SGI Altix 3700: 128x Itanium2 1.3 GHz	IT2+icc81	Intel C++ V8.1	9479.0	5182.8	fails

Table 1
Compilers and machines, runtime of original and tuned serial versions, OpenMP support.
Linux is running on all platforms, except for OPT+ss10, USIV+* (Solaris 10) and POW4+* (AIX).

disadvantage of this format is that insertion of a non-zero element into the matrix is rather expensive. Since this is unacceptable when building the matrix during the discretization step, a sparse matrix builder class has been designed with an intermediate storage format based on STL's `map` container that offers write access in logarithmic time for each element. After the assembly, the matrix is converted into the CRS format in the original program version.

3.4. Serial Tuning Measures

On the Opteron systems the PCG-algorithm including a sparse-matrix-vector-multiplication and the preconditioner profit from manual prefetching. The performance gain of the matrix-vector-

multiplication is 44% on average, and the speedup of the preconditioner is 19% on average, depending on the addressing mode (64bit mode profits slightly more than 32bit mode).

As the setup of the stiffness matrix turned out to be quite expensive, we reduced the usage of the `map` datatype. As long as the structure of the matrix does not change, we reuse the index vectors and only fill the matrix with new data values. This leads to a performance plus of 50% on the USIV+guide platform and about 58% on the OPT+icc platform. All other platforms benefit from this tuning measure as well.

Table 1 lists the results of performance measurements of the original serial version and the tuned serial version. Note that on the Opteron the 64bit addressing mode typically outperforms the 32bit mode, because in 64bit mode the Opteron offers more hardware registers and provides an ABI which allows for passing function parameters using these hardware registers. This outweights the fact that 64bit addresses take more cache space.

4. The OpenMP Approach

4.1. Assembly of the Stiffness Matrices

The routines for the assembly of the stiffness matrices typically contain loops like the following:

```
for (MultiGridCL::const_TriangTetraIteratorCL
   sit=_MG.GetTriangTetraBegin(lvl),
   send=_MG.GetTriangTetraEnd(lvl);
   sit != send; ++sit)
```

Such a loop construct cannot be parallelized with a `for`-worksharing construct in OpenMP, because the loop iteration variable is not of type integer. We considered three ways to parallelize these loops:

- The `for`-loop is placed in a parallel region and the loop-body is placed in a `single`-worksharing construct whose implicit barrier is omitted by specifying the `nowait`-directive. The problem with this approach is that the overhead at the entry of the `single`-region limits the possible speedup.

- Intel's compilers and the guidec++ compiler offer the task-queuing construct as an extension to the OpenMP standard. For each value of the loop iteration variable the loop-body is enqueued in a work-queue by one thread and then dequeued and processed by all threads. The number of loop iterations is rather high in relation to the work in the loop body, so again the administrative overhead limits the speedup. We proposed an extension of the task-queuing construct implemented by Intel for the upcoming OpenMP standard version 3 for which a schedule clause with a chunksize can be specified.

- The pointers of the iterators are stored in an array in an additional loop, so that afterwards a simpler loop running over the elements of this array can be parallelized with a `for`-worksharing construct. We found this approach to be the most efficient giving the highest speedup.

Reducing the usage of the `map` STL datatype during the stiffness matrix setup as described in chapter 3 turned out to cause additional complexity and memory requirements in the parallel version. In the parallel version each thread fills a private temporary container consisting of one map per matrix row. The structure of the complete stiffness matrix has to be determined, which can be parallelized over the matrix rows. The master thread then allocates the `valarray` STL objects. Finally, the matrix rows are summed up in parallel.

If the structure of the stiffness matrix does not change, each thread fills a private temporary container consisting of one `valarray` of the same size as the array *val* of the final matrix.

This causes massive scalability problems for the guidec++-compiler. Its STL library obviously uses critical regions to be threadsafe. Furthermore the guidec++ employs an additional allocator for small objects which adds more overhead because of internal synchronization. Therefore we implemented a special allocator and linked to the Sun-specific memory allocation library *mtmalloc* which is tuned for multithreaded applications to overcome this problem.

Thus, the matrix assembly could be completely parallelized, but the scalability is limited, because the overhead increases with the number of threads used (see table 2). The parallel algorithm executed with only one thread performs worse than the tuned serial version on most platforms, because the parallel algorithm contains the additional summation step as described above. On the USIV+guide platform it scales well up to about eight threads, but then the overhead which is caused by a growing number of dynamic memory allocations and memory copy operations increases. Therefore we limited the number of threads used for the SETUP routines to a maximum of eight in order to prevent a performance decrease for a higher thread count. On the USIV+ss10 and POW4+guide platforms there is still some speedup with more threads. Table 2 shows the runtime of the matrix setup routines. Note, that on the XEON-icc81 platform Hyper-Threading is profitable for the matrix setup.

code	**serial**	**serial**	**parallel**				
	original	**tuned**	**1**	**2**	**4**	**8**	**16**
XEON+icc81	1592	816	1106	733	577	n.a.	n.a.
OPT+icc81	1363	793	886	486	282	n.a.	n.a.
OPT+ss10	2759	1154	1233	714	428	n.a.	n.a.
USIV+guide	4512	2246	2389	1308	745	450	460
USIV+ss10	4564	1924	2048	1435	796	460	314
POW4+guide	4983	2236	2176	1326	774	390	185

Table 2
C++ + OpenMP: matrix setup, runtime [s]

4.2. The Linear Equation Solvers

In order to parallelize the PCG- and GMRES-methods, matrix and vector operations, which beforehand had been implemented using operator overloading, had to be rewritten with C-style `for` loops directly accessing the structure elements. Thereby some synchronizations could be avoided and some parallelized `for`-loops could be merged.

The parallelized linear equation solvers including the sparse-matrix-vector-multiplication scale quite well, except for the intrinsic sequential structure of the Gauss-Seidel preconditioner which can only be partially parallelized. Rearranging the operations in a blocking scheme improves the scalability (`omp_block`) but still introduces additional organization and synchronization overhead.

A modified parallelizable preconditioner (`jac0`) was implemented which affects the numerical behavior. It leads to an increase in iterations to fulfill the convergence criterium. Nevertheless it leads to an overall improvement with four or more threads.

The straight-forward parallelization of the sparse matrix vector multiplication turned out to have a load imbalance. Obviously the nonzero elements are not equally distributed over the rows. The load balancing could be easily improved by setting the loop scheduling to `SCHEDULE(STATIC,128)`.

The linear equation solvers put quite some pressure on the memory system. This clearly reveals the memory bandwidth bottleneck of the dual processor Intel-based machines (XEON+icc). The ccNUMA-architecture of the Opteron-based machines (OPT+icc) exhibits a high memory bandwidth if the data is properly allocated. But it turns out that the OpenMP version of DROPS suffers from the fact that most of the data is allocated in the master thread's memory because of the usage of the STL datatypes.

As an experiment we implemented a C++ version of the stream benchmark using the STL datatype `valarray` on one hand and simple C-style arrays on the other hand. These arrays are allocated with *malloc* and initialized in a parallel region. Table 3 lists the memory bandwidth in GB/s for one of the kernel loops (saxpying) and a varying number of threads. It is obvious that the memory bandwidth does not scale when `valarrays` are used. The master thread allocates and initializes (after construction a `valarray` has to be filled with zeros by default) a contiguous memory range for the `valarray` and because of the first touch memory allocation policy, all memory pages are put close to the master thread's processor. Later on, all other threads have to access the master thread's memory in parallel regions thus causing a severe bottleneck.

The Linux operating system currently does not allow an explicit or automatic data migration. The Solaris operating system offers the Memory Placement Optimization feature (MPO), which can be used for an explicit data migration. In our experiment we measured the kernels using `valarrays` after the data has been migrated by a "next-touch" mechanism using the *madvise* runtime function, which clearly improves parallel performance (see table 3). This little test demonstrates how sensitive the Opteron architecture reacts to disadvantageous memory allocation and how a "next-touch" mechanism can be employed beneficially. We proposed a corresponding enhancement of the OpenMP specification in the upcoming Version 3.0. On the USIV+guide, USIV+ss10 and OPT+ss10 platforms we were able to exploit the MPO feature of Solaris to improve the performance of DROPS.

Stream kernel	**Data structure**	**Initialization method**	**1 Thread**	**2 Threads**	**3 Threads**	**4 Threads**
saxpying	valarray	implicit	2.11	2.16	2.15	2.03
	valarray	implicit+madvise	2.10	4.18	6.20	8.20
	C-array	explicit parallel	2.15	4.26	6.30	8.34

Table 3
Stream benchmark, C++ (valarray) vs. C, memory bandwidth in GB/s on OPT+ss10

On the whole the linear equation solvers scale reasonably well, given that frequent synchronizations in the CG-type linear equation solvers are inevitable. The modified preconditioner takes more time than the original recursive algorithm for few threads, but it pays off for at least four threads. Table 4 shows the runtime of the solvers.

4.3. Total Performance

Table 5 shows the total runtime of the DROPS code on all platforms for which a parallel OpenMP version could be built. Please note that we didn't have exclusive access to the POW4 platform. Table 6 shows the resulting total speedup.

code	serial	serial	parallel (omp_block)				parallel (jac0)			
	original	tuned	1	2	4	8	1	2	4	8
XEON+icc81	939	894	746	593	780	n.a.	837	750	975	n.a.
OPT+icc81	901	835	783	541	465	n.a.	668	490	415	n.a.
OPT+ss10	731	673	n.a.	n.a.	n.a.	n.a.	596	375	278	n.a.
USIV+guide	2682	2727	2702	1553	1091	957	1563	902	524	320
USIV+ss10	2714	2652	2870	1633	982	816	2555	1314	659	347
POW4+guide	440	441	589	315	234	159	572	331	183	113

Table 4
C++ + OpenMP: linear equation solvers, runtime [s]

code	serial	serial	parallel (omp_block)				parallel (jac0)				
	original	tuned	1	2	4	8	1	2	4	8	16
XEON+icc81	2643	1723	2001	1374	1353	n.a.	2022	1511	1539	n.a.	n.a.
OPT+icc81	2404	1767	1856	1233	952	n.a.	1738	1162	891	n.a.	n.a.
OPT+ss10	3613	2431	n.a.	n.a.	n.a.	n.a.	1973	1234	856	n.a.	n.a.
USIV+guide	7551	5335	5598	3374	2319	1890	4389	2659	1746	1229	1134
USIV+ss10	7674	4959	5422	3573	2255	1736	5056	3214	1894	1250	956
POW4+guide	5535	2790	3017	1752	1153	655	2885	1084	1099	641	482

Table 5
C++ + OpenMP: total runtime, runtime [s]

code	serial	serial	parallel (omp_block)				parallel (jac0)				
	original	tuned	1	2	4	8	1	2	4	8	16
XEON+icc81	1.00	1.53	1.32	1.92	1.95	n.a.	1.31	1.75	1.72	n.a.	n.a.
OPT+icc81	1.00	1.36	1.30	1.95	2.53	n.a.	1.38	2.07	2.70	n.a.	n.a.
OPT+ss10	1.00	1.49	n.a.	n.a.	n.a.	n.a.	1.83	2.93	4.22	n.a.	n.a.
USIV+guide	1.00	1.42	1.35	2.24	3.26	3.99	1.72	2.84	4.32	6.14	6.66
USIV+ss10	1.00	1.55	1.42	2.15	3.40	4.42	1.52	2.39	4.05	6.14	8.03
POW4+guide	1.00	1.98	1.83	3.16	4.80	9.73	1.92	3.07	5.04	8.63	11.48

Table 6
C++ + OpenMP: speed-up

5. Summary

The compute intense program parts of the DROPS Navier-Stokes solver have been tuned and parallelized with OpenMP. The heavy usage of templates in this C++ program package is a challenge for many compilers. As not all C++ compilers support OpenMP, and some of those which do, fail for the parallel version of DROPS, the number of suitable platforms turned out to be limited.

We ended up with using the guidec++ compiler from KAI (which is now part of Intel) and the Sun Studio 10 compilers on our UltraSPARC IV-based and Opteron-based Sun Fire servers and the Intel compiler in 32 bit mode on our Opteron-based Linux cluster.

The strategy which we used for the parallelization of the Finite Element Method implemented

in DROPS was straight forward. Nevertheless the obstacles which we encountered were manifold, many of them are not new to OpenMP programmers.

OpenMP programs running on big server machines operating in multi-user mode suffer from a high variation in runtime. Thus it is hard to see clear trends concerning speed-up. Exclusive access to the UltraSPARC IV-, Opteron- and Xeon-based systems helped a lot. On the 4-way Opteron systems the *taskset* Linux command was helpful to get rid of negative process scheduling effects.

We obtained the best absolute performance and the best parallelization speed-up on the POW4+guide platform, using a compiler which is no longer available. The best OpenMP version runs 11.5 times faster with 16 threads than the original serial version on the same platform (POW4+guide). But as we improved the serial version during the tuning and parallelization process, the speed-up compared to the tuned serial version is only 5.8.

On the OPT+ss10 platform, the best OpenMP version runs 4.2 faster than the original serial version and 2.8 faster than the tuned serial version with four threads.

An Opteron processor outperforms a single UltraSPARC IV processor core by about a factor of four. As Opteron processors are not available in large shared memory machines and scalability levels off with more than sixteen threads, shorter elapsed times are currently not attainable.

Acknowledgements
The authors would like to thank Uwe Mordhorst at University of Kiel and Bernd Mohr at Research Center Jülich for granting access to and supporting the usage of their machines, an SGI Altix 3700 and an IBM p690, respectively.

References

[1] Sven Gross, Jörg Peters, Volker Reichelt, Arnold Reusken: The DROPS package for numerical simulations of incompressible flows using parallel adaptive multigrid techniques. ftp://ftp.igpm.rwth-aachen.de/pub/reports/pdf/IGPM211_N.pdf
[2] Arnold Reusken, Volker Reichelt: Multigrid Methods for the Numerical Simulation of Reactive Multiphase Fluid Flow Models (DROPS). http://www.sfb540.rwth-aachen.de/Projects/tpb4.php
[3] GNU Compiler documentation: http://gcc.gnu.org/onlinedocs/
[4] Intel C/C++ Compiler documentation: http://support.intel.com/support/performancetools/c/linux/manual.htm
[5] PGI-Compiler documentation: http://www.pgroup.com/resources/docs.htm
[6] Pathscale-Compiler: http://www.pathscale.com
[7] Sun Studio 10: http://developers.sun.com/prodtech/cc/documentation/ss10_docs/
[8] Guide-Compiler of the KAP Pro/Toolset: http://developer.intel.com/software/products/kappro/

Optimization of an octree-based 3-D parallel meshing algorithm for the simulation of small-feature semiconductor devices

Juan J. Pombo[a], M. Aldegunde[a], A.J. Garcia-Loureiro[a]

[a]Department of Electronics and Computer Science, University of Santiago de Compostela 15782 - Santiago de Compostela (Spain)

1. Abstract

This paper describes an optimized meshing strategy that fits the requirements of a finite element (FEM) simulation software for small-feature semiconductor devices. The main characteristic of these devices is the existence of very small layers of material and transiction regions where several magnitudes of interest can change very abruptly. Because of that, elements in the mesh for the 3D FEM have to be be positioned quite carefully in order to fulfil different size and vertex position requirements. In other hand, as a consecuence of the small details in the devices that should be captured, the number of elements can be moderately high, while geometries, however, are not very complex. In this scene an adapted parallel octree for volume representation, combined with a pattern based algorithm that embeds the tetrahedral mesh in the octree is shown as a feasible solution that performs sucessfully in a parallel computer.

2. Introduction

In the context of the numerical simulation of new semiconductor devices efforts are being made to achieve light-weigthed tools that model correctly the behaviour of small-feature devices. Several kinds of common new transistors are approaching nowadays nanometric features. So is the case of the submicron high electron mobility transistors (HEMT), submicron MOSFETs, bipolar junction transistors (BJT) or heterojunction bipolar transistors (HBT).

One of the difficults that arise in this kind of problems is the very elongated models to consider. The total lenght dimensions in Y axis could be as much as twenty times those on the X axis.

These devices also may present very thin layers of different materials or doping characteristics. In many cases corresponding also to different materials. So nodes of the mesh must be positioned exactly in the boundary.

As a consequence this situation demands special care to guarantee certain quality criteria [9]. The simulation process have to deal with the resolution of equations over very small or stretched regions of materials. In order to guarantee that Finite Element Methods (FEM) based analysis produces adequate results, tetrahedral meshes with special grading control and form factors have to be created. In other case, the meshing process could result in an enormous amount of small elements that causes the performance of the whole simulator to decay drastically.

Modelling the behaviour of these small-geometry devices is very important to develop CAD tools that help to the adjustment of device parameters in the design process [8], [7]. This work describes the efforts to obtain a reliable 3D meshing tool as a part of a complete parallel simulation software in development in our group [6]. The simulation program is able to work well with distributed data, generating distributed tetrahedral meshes suitable for a parallel Finite Element Method (FEM) solver to proceed without important data modifications.

In the present work we focus our attention just in the algorithm for the meshing part of the pro-

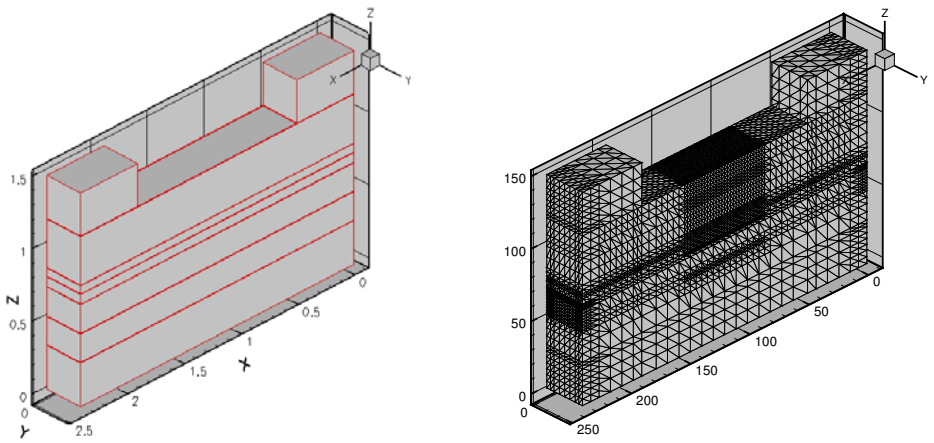
cess. However, the meshes obtained have been tested to ensure that they are suitable for a correct numerical simulation.

The meshing algorithm presented take advantage of the use of a parallel octree based strategy and is able to overcome sucesfully several difficulties related to this kind of problems. It performs adecuately, presents scalability and is suitable for a correct numerical simulation.

3. The semiconductor problem

In order to adapt the resulting mesh to the requirements of the problems at hand some constraints have been considered in the program. In the next paragraphs we summarize and shortly describe some of them.

Multilayer conformity: The problems that we consider in this paper do not present boundaries with very complex geometries and most of the faces are well represented with planar configurations. This is quite common in this kind of simulations and we take advantage of it in several steps. However, new difficulties are caused by the big amount of planar layers existent, some of them extremely close to the others (Figure 1). In case of multilayer structures node occurrences become mandatory in the boundary between different materials. These boundaries are represented as inner faces and have a special consideration across the meshing program.



(a) (b)

Figure 1. (a) Solid model for a HEMT structure and (b) final tetrahedral mesh

Minimized shared mesh: For the distributed tetrahedral mesh, an overlapping must exist between the local meshes assigned to each processor and the “offprocesor” regions (as required by the numerical solver). This fact makes necessary to minimize the amount of elements in the shared region between processor. In other case it would cause degradation of the scalability and the whole performance of the solution process. See Section 5.1 for more details.

Refinement representation: Depending on the problem, different models for representing the level of refinement in the mesh are necessary. Most common refinement criteria are achieved by forcing a constrained size of element along faces or general planes in the 3D structure. However some

more complex criteria are necessary in those cases where general functions describing doping are introduced. The octree strategy requires a special treatment to incorporate these requirements at the octree level.

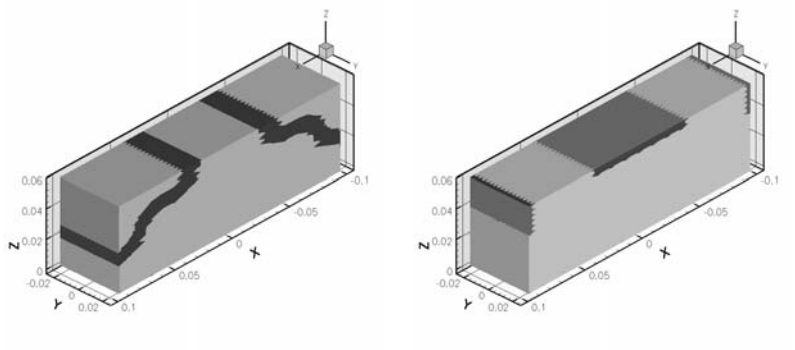


Figure 2. Information of refinement for a MOS transistor. Contacts and other regions of maximum refinement are marked.

Limited mesh size. A compromise exists between good convergence of the FEM problem over the mesh and a reasonable number of elements. It has no sense just increase elements to solve with better accuracy. In 3-D simulators the computations increase enourmously with the number of nodes in the mesh. A correct graded control of the refinement introduced in the last paragraph is a key for preserving good overall performance. This objective is not trivial in an octree-based approach. The thecnique presented here is based in a "default" level of refinement combined with control points.

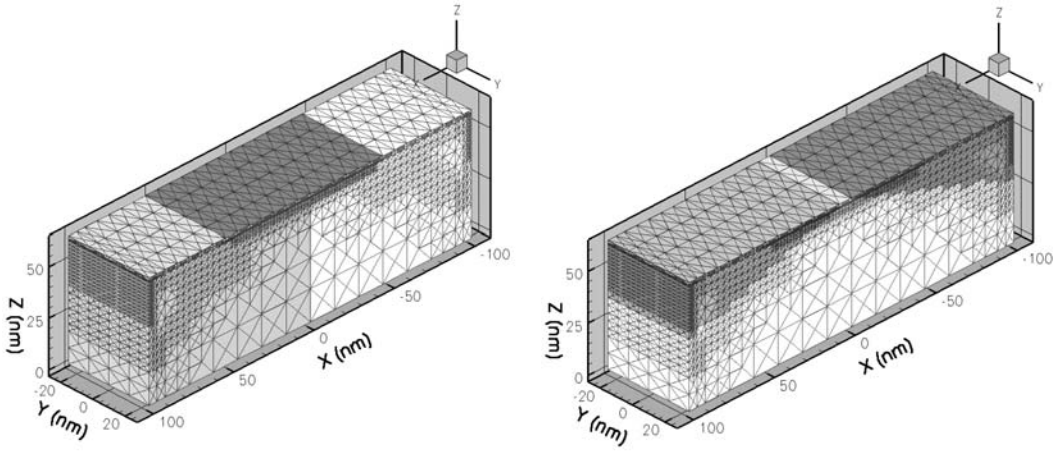
Scaling capability. Very often the elongated sizes in some dimensions of the small-feature transistors make necessary some support for scaling the models previously to the meshing process. This process is desirable for the good performance of the octree based mesher, since it avoids umbalanced trees because of the numerous tree branches outside the solid model. However, the effects of these approaches on the form factor of the tetrahedra establish a limit to this technique when convergence criteria are taken into account. Constraints based on form factors have been introduced.

Automatic load balancing. Parallel generation of the octree is intended to guarantee an adequate load balance of the computing work among processors while minimizing global communications that could degrade algorithm performance. The distribution resulting from this automatic process will be compared to the case of an optimization using mesh partitioning tools like [5].

4. Parallel algorithm

4.1. The octree-method

Octree methods are based on hierarchical space decom position [1]. The process starts constructing an initial cuboid enclosing the whole region. This octant is recursively subdivided into eight parts. The subdivision continues until some refinement criteria are achieved. Only those octants intersecting the boundary of the region or totally enclosed by it are taken into account for further subdivision. Refinement criteria are introduced during the aforementioned process as an input containing a global minimum level of the octree and local specifications. Those criteria are introduced by mean of control points indicating the required octree level in each region (see Figure 3). The goal of reaching these levels guides the creation of the octree.



(a) (b)

Figure 3. Two different mesh distributions for 3 processors in a MOS problem

The use of this data structure allows an easy way to manage the storage of the information needed during the meshing process. Operations which use this information include neighbour finding and boundary adaptation. Both are fundamental parts in the generation of the octree to be meshed. Specially important is the first one, which will be accessed thousands of times (or even millions in case of large meshes) during mesh generation. A description of the neighbour finding algorithm implemented can be found in [4].

4.2. Architecture of the parallel program

We can obtain several benefits from the use of the octree as a model representation. First, the geometric model at hand is subdivided and partitioned in a natural way. The tree is generated in parallel. We take advantage of this situation to divide the load of work among the processors without overheads. Octants of level 0 (root), 1, and 2 are created equal in all the processors. The 64 branches of level 2 in the *octree* are distributed in blocks (Figure 3). In each processor exist only the branches of the octree assigned to itself. In order to access other branches considered "off-processor" in a traversal operation, there exist links in certain octants called *gateways* that allow a fast tracking down of the tree.

The data structure that results, the octree, ensures also a hierarchical and easy to use storage of the information needed during the meshing process. Searching operations across large data arrays are avoided when possible. On the other hand, several algorithms are implemented taking into account the special properties of the octree. We have to be specially careful with the selected data distribution, mainly when we work with a parallel approach to the searching operation. The main kernel that has to be implemented over the octree is the vicinity finding operation. Efforts have been made to code these procedures in an efficient way, since a lot of vicinity tests, about shared faces or vertices between entities, have to be completed during the process. And these tests involve, sometimes, information associated to other processors.

Figure 4 depicts this point. Starting in the most refined leave nodes of the tree somekind of advancing front is created growing in the local octree based on vicinity operations. Visited leave nodes that are not sufficiently refined are partitioned, and neighbourg octants not local to the processor are annotated and sent to the processors that can solve them. This is a one way communication that

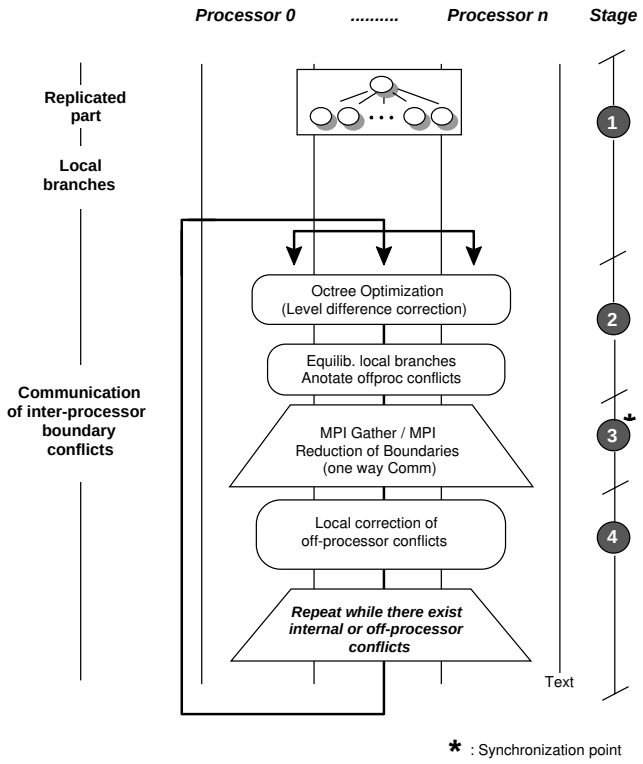


Figure 4. *Octree* construction scheme

do not require response. Each processor then solves their the problem locally (steps 2, 3 and 4 in Figure 4.

5. Tetrahedral mesh generation

Once we have the octree adapted to the geometry, layer constraints and refinement criteria an algorithm to embed the tetrahedral mesh based on the templates method is executed. It is possible to use several algorithms for generating the tetrahedra. The algorithm we select depends on the requirements of the situation. Sometimes we want a faster execution with similar resolution (given by the octree) and sometimes we need a mesh with elements of higher quality. These different methods available are based on the same underlying idea. We generate a basic set of nodes for each terminal octant, and then connect them using a well stablished pattern (Figure 5). We consider in this paper a case where eight vertex are generated for a normal octant. New extra vertices (and elements) could appear in the centers of octant, face or edge, taking into consideration more refined neighbours, and assuring conformality of the mesh.

This strategy, applied to semiconductor devices, reduces the number of nodes and elements in mesh. The only problem for parallelism is that these “extra” nodes are always taken from the most refined neighbours, and sometimes these octants belong to other processor. It seems not feasible just to communicate between processors to obtain the needed coords of the shared node each time the problem appears during the local octree traversal. If we did that the flux of the program would be continuously interrupted. Instead of that, our algorithm creates a structure that solves the eighteen posible directions of vicinity, and is prepared in advance. Creating this structure implies communication with other processors, finding of remote neighbourgh octants, and return of auxiliar coordinates.

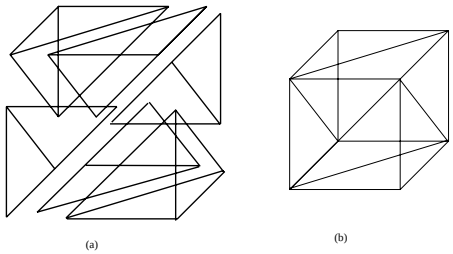


Figure 5. Tetrahedral pattern.

These stages are noted as 2,3 and 4 in Figure(6). It also involves a slight memory overhead. It should be noted that, even though this stages are not the most time consuming steps, they are very costly in terms of scalability. In Section 6 this effect is studied in detail.

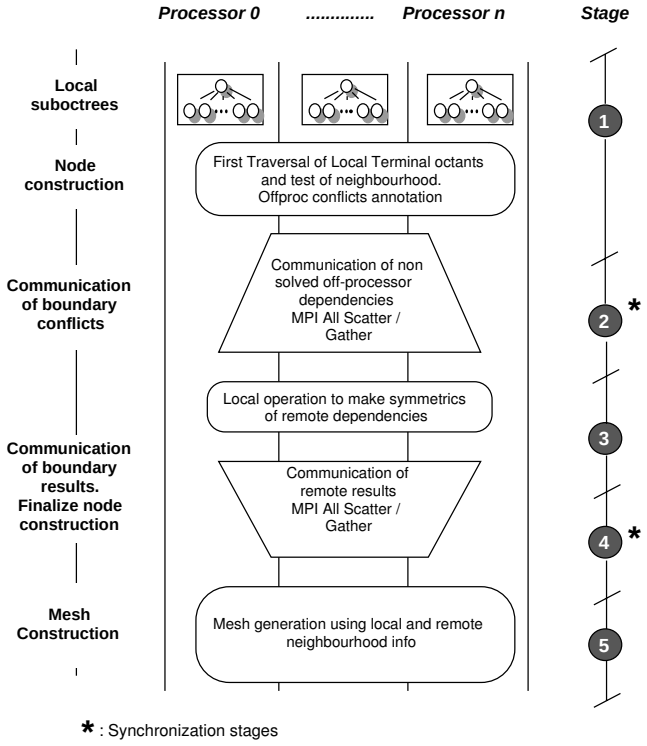


Figure 6. Tetrahedralization scheme

5.1. Analisis of the communication process

Communication between processors occurs in this implementation in several stages along the complete execution. Tuning these stages is very important to get good performance. In any of these steps, when executed with a low number of processors, it is very feasible that communications “all-to-all” could be necesary, since most of the processors can contain neighbour branches of the *octree*. Fortunally, as the number of processors increases, this behavior reachs a limit allowing reasonable scalability.

Since the aforementioned communication stages introduce some kind of syncronization, an ade-

quate load balance is very important. The difficulties to achieve this balance come from the fact that the exact number of *terminal* octants (leave nodes of the *octree*) is only known after obtaining the complete tree, since the generation of the *octree* is completed in a distributed mode. So, the offspring of distributable octant has to be estimated at the beginning of the process, using the refinement control points. When using the mesher in an adaptive scheme [3], this problem is not so important, since changes in the *octree* distribution could be done at the beginning of each new iteraction.

6. Results and Performance

The code has been developed using MPI [2]. It has been executed on a Silicon Graphics Origin 200 and a cluster HP Integrity Superdome. The results are shown in Table 1 and Table 2. A load balance resume is also presented in Table 3.

The labels in the columns of the tables of execution times must be read as follows: OCTGEN, octree generation time. ADJ, time used to adjust octants to boundaries and layers. OPT, process of optimizing the level difference between branches of the *octree* (see Figure 4). TOTAL OCT, total octree generation time. COM(2WAYS), time used for solving neighbourhood offprocessor, there are two steps, send problems, and return of results, (see Figure 6). LOCSIMET, process in the middle of the previous ones, used to compute the problems of offprocessor neighbourhood that can be managed locally. TMESH, complete time of the element generation. TOT T, time used by the whole process.

We can obtain several conclusions checking these results. First of all, it is noticeable that the most consuming part of the algorithm is the preparation of the *octree* structure. The good point is that this part scales very nicely. Once the octants are adjusted in such a manner that neighbour octants differ at most in one level of refinement, defining the tetrahedral elements and nodes involved is a straightforward procedure. This part presents lower scalability but it supposes a small percentage of the complete meshing time.

Load balancing is good enough for not having negative influences on performance.

Scalability is good for small number of processors, but freezes in certain point. This is not such a negative behaviour taking into account the very irregular code that we are analyzing. The key point is that, given a dimension of a problem, we have a limit in the number of processors in which we can obtain important speedups. If the size of the problem increases it is likely that scalability reappears.

We have shown in the tables a quite small problem, in terms of number of terminal octants. However, it makes sense to consider it, since in a FEM simulation based on an adaptive loop the meshing process can be quite time consuming if the number of interactions increase, and mainly when this meshing process is sequential, and data of the mesh has to be partitioned and redistributed in each loop.

In the superdome the effect of the speedy CPU's decreases the scalability. Values of speedups greater than the *np* number appear in the case of the Origin machine. This is not an effect of superscalability, but a cause of the different behaviour in the algorithm when it is executed in parallel, since it is intrinsically parallel.

References

[1] Mark A. Yerry and Mark S. Shephard. Automatic Three-dimensional mesh generation by the Modified-Octree Technique. Int. J. for Num. Methods in Eng. Vol. 20, pp.1965-1990, 1984.

[2] Message Passing Interface Forum. MPI: A Message Interface Standard. Computer Science Department, University of Tennessee. CS-94-230, Knoxville, TN, 1994.

[3] M.S. Shephard, J.E. Flaherty, C.L. Bottasso, H.L. de'Cougny, C. Ozturan, M.L. Simone. Parallel auto-

Execution Times in Origin 200(secs)									
NProc	Execution Time (secs)								**Speedup**
	OCTGEN	ADJ	OPT	Total OCT	COM(2WAYS)	LOCSIMET	TMESH	TOT T	
1	5.05	1.64	0.18	7.9			1.51	9.51	
2	2.67	0.79	0.09	3.55	0.05	0.11	0.16	3.71	2.56
3	2.04	0.62	0.08	2.74	0.34	0.02	0.36	3.10	3.07
4	1.41	0.43	0.12	1.96	0.28	0.03	0.31	2.2	4.32

Table 1
Execution Times for the HEMT transistor in the Origin 200

Execution Times in HP Superdome									
NProc	Execution Time (secs)								**Speedup**
	OCTGEN	ADJ	OPT	Total OCT	COM(2WAYS)	LOCSIMET	TMESH	TOT T	
1	1.02	0.33	0.02	1.37			0.23	1.6	
2	0.56	0.16	0.05	0.77	0.01	0.03	0.12	0.89	1.79
3	0.32	0.10	0.12	0.54	0.06	0.02	0.08	0.62	2.58
4	0.24	0.08	0.08	0.4	0.04	0.02	0.06	0.46	3.58
5	0.20	0.06	0.07	0.33	0.04	0.03	0.07	0.40	4.0
6	0.20	0.06	0.08	0.34	0.05	0.02	0.07	0.40	4.0

Table 2
Execution Times for the HEMT transistor in Cluster HP Integrity Superdome

Load Balance for the HEMT mesh	
Number of processors	**Number of Terminal octants**
1	21384
2	10748/10636
3	6342/5470/5318
4	5318/5374/5238/5264
5	2240/4278/4326/5222/5086
6	2088/2088/4406/4174/4430/3966

Table 3
Number of terminal octants meshing the HEMT problem

matic adaptive analysis. Parallel automatic adaptive analysis. Parallel Computing, Vol. 23, pp.1327-1347, 1996.

[4] J. J. Pombo, T. F. Pena and J. C. Cabaleiro. Parallel Complete Remeshing for Adaptive Schemes. Proc. HPSECA-ICPP, 2001.

[5] G. Karypis and V. Kumar. METIS: A software Package. The University of Minnesota. 1997.

[6] A.J.Garcia-Loureiro, K. Kalna and A. Asenov. 3D Parallel Simulations of Fluctuation Effects in pHEMTs. J. Comp. Elec, Vol.2, pp.369-373, 2003

[7] QMG 1.1 Reference Manual. Computer Science Departament, Cornell University, 1996.

[8] B. H. V. Topping, J. Muylle, P. Ivany, R. Putanowicz and B. Cheng. Finite Element Mesh Generatio, Saxe-Coburg Publications. Kippen, Stitrling, Scotland, 2004.

[9] N. Hitschfeld and M. C. Rivara. Improving the quality of meshes for the simulation of semiconductor devices using Lepp-based algorithms. Int. Journ. for Num. Methods in Enginnering. Vol.5, pp.333-347, 2003.

Large Scale Simulation of Ideal Quantum Computers on SMP-Clusters

G. Arnold[a], Th. Lippert[a], N. Pomplun[a] and M. Richter[a]

[a]Central Institute for Applied Mathematics, Research Centre Juelich, 52425 Juelich, Germany

1. Relevance of Quantum Computing

Quantum processing of information has become a rapidly evolving field of research in physics, mathematics, computer science, and engineering [1] and has led to substantial progress in quantum computation, quantum communication and control of quantum systems. Quantum computers have become of great interest primarily due to their potential of solving certain computationally hard problems such as factoring integers [2] and searching databases faster than a conventional computer [3]. Candidate technologies for realizing quantum computers include trapped ions, atoms in QED cavities, Josephson junctions, nuclear or electronic spins, quantum dots, and molecular magnets. Grover's quantum search [3] and Shor's quantum prime factorization algorithm [2] have been successfully implemented on systems of up to 7 qubits using liquid NMR techniques [4], experimentally demonstrating the viability of the concept of quantum computation.

In spite of this impressive development, a demonstration that quantum computation can solve a non-trivial problem is still lacking. To be of practical use, quantum computers will need error correction, which requires at least several tens of qubits and the ability to perform hundreds of gate operations. This imposes a number of strict requirements [5], and narrows down the list of candidate physical systems. Simulating numbers of qubits in this range is important to numerically test the scalability of error correction codes and fault tolerant quantum computing schemes and their robustness to errors typically encountered in realistic quantum computer architectures.

2. The Need for Simulation

A physically realizable quantum computer is a complex many-body quantum system. In order to exercise control over many qubits and to suppress the rate at which errors are introduced during a quantum computation, it is in principle necessary to understand the full time evolution of the whole quantum system. Sources of errors are the loss of coherence (decoherence) due to unwanted interaction with the environment [6] and systematic errors due to imperfections of the operational pulse sequences.

In *first principle* simulations the time dependent behavior can be derived from the Hamiltonian of the physical model chosen to describe a specific hardware realization. Pulses are modeled as time-dependent external fields acting on the relevant degrees of freedom. The coupling of the environment is taken into account by including interactions with other degrees of freedom, also represented by pseudo-spins.

These kind of simulations are needed to analyze decoherence resulting from interactions with the environment. Depending on the assumptions that where made in deriving the microscopic Hamiltonian and/or the manner in which the effect of the coupling to the environment is taken into account, the calculation of the real-time quantum dynamics of the quantum computer readily requires the simulation of systems of many (20-40) qubits over extended periods of time. To perform such very demanding computations, highly optimized simulation code that runs on different high-end computer systems has to be developed.

3. Simulation of Ideal Quantum Computers

In a first step towards realistic quantum computer simulations we implement so called *ideal simulations*, where each gate is modeled by a quantum operation that acts instantaneously on the internal state of the quantum computer, neglecting both implementation imperfections and interactions with the environment. The drawback is that the state of the quantum computer is known only after the application of each gate, but this is sufficient for most algorithmic purposes.

In contrast to a classical bit the state of an elementary storage unit of a quantum computer, the quantum bit or qubit, is described by a two-dimensional vector of Euclidean length one. Denoting two orthogonal basis vectors of the two-dimensional vector space by $|0\rangle$ and $|1\rangle$, the state $|\psi\rangle$ of a **single qubit** can be written as a linear superposition of the basis states $|0\rangle$ and $|1\rangle$:

$$|\psi\rangle_1 = a_0|0\rangle + a_1|1\rangle \ = \begin{pmatrix} a_0 \\ a_1 \end{pmatrix}, \tag{1}$$

where a_0 and a_1 are complex numbers such that $|a_0|^2 + |a_1|^2 = 1$. Useful computations require more than one qubit. The state of a quantum computer with N qubits can be represented in the 2^N-dimensional Hilbert space as

$$\begin{aligned} |\psi\rangle_N &= a_{0\ldots00}|0\ldots00\rangle + a_{0\ldots01}|0\ldots01\rangle + \ldots + a_{1\ldots10}|1\ldots10\rangle + a_{1\ldots11}|1\ldots11\rangle, \\ &= a_0|0\rangle + a_1|1\rangle + \ldots + a_{2^N\ \ 1}|2^N-1\rangle \\ &= (a_0, a_1, \ \ldots \ , a_{2^N\ \ 1})^T . \end{aligned} \tag{2}$$

According to the rules of quantum mechanics any evolution in time means changing the system state unitarily. Each operation on a quantum computer can be described by a 2^Nx 2^N dimensional unitary transformation $U = e^{\ iHt}$ acting on the state vector $|\psi'\rangle = U|\psi\rangle$, with the hermitian matrix H being the Hamiltonian of the quantum computer model. In this paper we will not describe any details of quantum computer hardware modeled by appropriate Hamiltonians. It is sufficient to know that an ideal quantum computer can be modeled by simple spin models such as the Ising model associating the two single-spin states *up*$= |\uparrow\rangle$ and *down*$=|\downarrow\rangle$ with the single-qubit basis states $|0\rangle$ and $|1\rangle$ [7].

As the unitary transformation U may change all amplitudes simultaneously, a quantum computer is a massively parallel machine. In order to simulate an arbitrary unitary operation on a conventional computer the resulting matrix-vector multiplication requires in the worst case $\mathcal{O}(2^{2N})$ complex valued arithmetic operations.

As in the case of programming a conventional computer, it is extremely difficult to write down explicitly that single one-step operation that transforms the input state into a desired output state. Usually a quantum algorithm consists of a sequence of many elementary gates. These elementary gates are represented by very sparse unitary matrices. The resultant matrix-vector multiplication can be implemented very efficiently and requires typically $\mathcal{O}(2^N)$ arithmetic operations per elementary gate. A small set of elementary **one-qubit gates** (such as the Hadamard gate and the Phase shift gate) and a nontrivial **two-qubit gate** (such as the controlled NOT gate) are sufficient (but not unique) to construct a *universal* quantum computer [8]. In the framework of ideal quantum operations any one-(two-) qubit operation can be decomposed into a sequence of 2x2 (4x4) matrix operations each acting on an orthogonal subspace of the 2^N dimensional Hilbert space.

In the following we describe an efficient parallel simulation of ideal quantum computers on a high-end computer system that allows simulating up to 37 qubits requiring 3 TB of memory and a considerable compute power.

4. One-Qubit Operations

We will discuss in detail the implementation of a typical one-qubit operation, the Hadamard gate. This gate is often used to prepare the state of uniform superposition. The Hadamard operation on a single-qubit state is defined by

$$\begin{aligned} |0\rangle &\rightarrow \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \\ |1\rangle &\rightarrow \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle). \end{aligned} \qquad H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}.$$

Let us consider a quantum computer consisting of three qubits and its state vector $|\psi\rangle = (a_{000}, a_{001}, a_{010}, a_{011}, a_{100}, a_{101}, a_{110}, a_{111})^T$. Instead of computing the 8x8 matrix appropriate to a Hadamard operation H_q acting on qubit q we can compute $H_q|\psi\rangle$ as given by the scheme in Fig. 1.

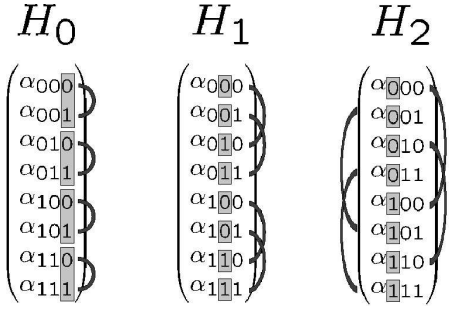


Figure 1. Decomposing a Hadamard transformation acting on qubit q on a three qubit computer $H_q|\psi\rangle$ into four parallel applications of the single-qubit Hadamard gate H. H_0 for example splits into $\binom{a_{ij0}}{a_{ij1}} \mapsto H\binom{a_{ij0}}{a_{ij1}}$ with $i, j \in \{0, 1\}$.

From this simple example we can learn some important (generalizable) characteristics for the Hadamard transformation H_q on a N-qubit quantum computer influencing qubit $q = 0, \ldots, N-1$ by acting on the 2^N- dimensional state vector $(a_{0\ldots00}, a_{0\ldots01}, \ldots, a_{1\ldots11})^T$:

i) H_q can be decomposed into 2^{N-1} applications of H involving a pair of state vector components (k, l) with relative stride $|l - k| = 2^q$ each.

ii) all these matrices H operate on orthogonal subspaces of the 2^N- dim Hilbert space. Hence they commute and computations can be done in parallel.

From i) we can derive that with the exception of H_0 all Hadamard gates operate purely on even *or* odd state vector elements. This claim also holds for any other quantum operation that does not involve qubit 0. Thus we will split the state vector $|\psi\rangle$ given by Eq.(2) into an even part $|\psi_e\rangle$ and its odd counterpart $|\psi_o\rangle$. For $0 \quad k \quad 2^{N-1} - 1$ we define:

$$|\psi_e(k)\rangle = |\psi(2k)\rangle \quad \text{and} \quad |\psi_o(k)\rangle = |\psi(2k+1)\rangle. \tag{3}$$

This state vector splitting saves half the effort to determine all pairs of indices (k, l) involved in the corresponding one-qubit operation. For H_q with $q > 0$ consecutive pairs (k, l) are mapped to identical pairs $(k', l')_e = (k', l')_o$ with stride $|l' - k'| = 2^{q-1}$. Only H_0 shows an even and odd mixing but trivial pattern that is implemented differently.

5. Parallelization and Computational Resources

More important than gaining half of the integer arithmetics, needed for state vector referencing, the splitting decreases communication overhead in the parallelized simulation code due to sending and receiving *non-stridden* sections of the state vectors $|\psi_e\rangle$ and $|\psi_o\rangle$. This leads to a gain of up to 30% of the wall clock time for one-qubit operations (depending on the system size N and the qubit number q the gate operates on). In this section we present some parallelization details.

The problem of simulating quantum computers is clearly memory bounded. Due to the exponentially increasing amount of memory needed we developed and implemented a large scale simulation on the Juelich SMP supercomputer IBM p690 providing enough memory to handle a 37 qubit system. Simple storage of the state vector in case of a 37 qubit system requires a memory of 2 TB. An efficient implementation of quantum operations on that state vector even requires 3 TB of memory.

The Juelich IBM p690 is a *cluster of 32 compute nodes each containing 32 Power 4+ processors* (64bit) and 112 GB memory per node leading to 3.5 TB overall memory available to user access. Two processors share a L2 cache of 1.5 MB and each node shares a 512 MB L3 cache. Users normally only have access to max. 16 nodes equivalent to 512 processors.

A quantum computer with up to $N = 32$ qubits reserving at max. $2^{36} = 64$ GB memory to store the complex valued state vector in double precision can be simulated on one node using 32 processors. In the following table we describe the typical simulation requirements depending on the system size N. The last row indicates the overall memory requirements to efficiently simulate quantum operations including the amount of memory to store the state vector.

#qubits N	32	33	34	35	36	*37*
#procs	32	64	128	256	512	*1024*
#nodes	1	2	4	8	16	*32*
memory (state vector)	64 GB	128 GB	256 GB	512 GB	1 TB	*2 TB*
memory (operation)	96 GB	192 GB	384 GB	768 GB	1.5 TB	*3 TB*

Partitioning the 2^N- dimensional state vector into $P = 2^p$ *tasks* allows to store the state vector of a N-qubit quantum computer on 2^{N-32} compute nodes equivalent to 2^{N-27} processors with the obvious limitation

$$N - 32 \quad p \quad N - 27. \tag{4}$$

MPI-based exchange of the local $|\psi_e\rangle$ and $|\psi_o\rangle$ allows computation of one-qubit operations on "nonlocal" qubit $q \quad N - p$. In that case the relevant state vector components (k, l), the single-qubit gate operates on, are separated as wide as (or wider than) the number of states per task: $|l - k| = 2^q \quad 2^{N-p}$. As shown in Fig.2 task K (containing all components k) sends its local $|\psi_e\rangle_K$ to task L and receives the local part $|\psi_o\rangle_L$ from task L.

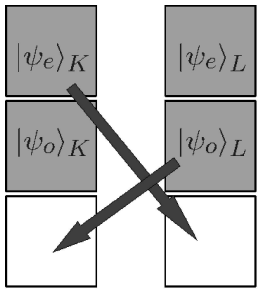


Figure 2. Communication pattern for a one-qubit operation on qubit $q > N - p$. The N qubit state vector is partitioned into 2^p tasks. The computational effort is equally distributed to $2^{p\ 1}$ disjoint pairs of tasks (K, L). Task K (L) operates on all odd (even) state vector elements $|\psi_o\rangle_{K+L}$ $(|\psi_e\rangle_{K+L})$.

After task K has computed locally the operation $H|\psi_o\rangle_{K+L}$ on all $(k, l)_o$ and task L has computed the even part $H|\psi_e\rangle_{K+L}$ on all $(k, l)_e$ respectively, both send back their results: K sends $H|\psi_o\rangle_L$ to L and receives $H|\psi_e\rangle_K$ from L. After that K contains the updated vectors $|\psi'_{e/o}\rangle_K = H|\psi_{e/o}\rangle_K$. L respectively stores $|\psi'_{e/o}\rangle_L = H|\psi_{e/o}\rangle_L$ in place. Thus the operation requires an intermediate buffer of $2^{N\ p\ 1}$ elements (half the size of the local state vector). Remember that any one-qubit operation on qubit 0 is local.

- Setting p to the maximum given by Eq.(4) *(finest graining)* means distributing the state vector on all processors of the nodes involved. This is equivalent to a pure MPI parallelization ansatz. For data exchange within a node the MPI-library is mapped to fast shared memory access.
- Choosing the minimal number of tasks given by Eq.(4) *(coarsest graining)* leads to one task per node which means that 32 processors are available to that task in parallel. To do this the core routine (a double loop) is done in parallel by $T = 2^t = 32$ OpenMP threads using shared memory access to the whole node memory of 64 GB reserved for the "local" state vector.
- Any other choice of $p + t = N - 27$ with $t \quad 0$ leads to an a priori reasonable *hybrid* parallelization strategy in the sense that all processors of the involved nodes are used for computation.

Our detailed investigation on systems of sizes $N = 32, 33, 34, 35, 36$ shows that different OpenMP parallelization strategies using more than 4 threads per MPI-task fail to reach the efficiency of the pure MPI-parallelization. Since we cannot provide a large enough additional buffer, task K for example is forced to operate "in place" on the state vector parts $H|\psi_o\rangle_{K+L}$ having write access to a *global=shared* vector. In that case synchronization of different OpenMP threads becomes necessary (within a task) and slows down computation.

Finest graining in the pure MPI-ansatz benefits from simple coding of one qubit operations on nonlocal qubits $q \quad N - p$ with maximal p according to Eq.(4). Since the stride $|l - k|$ is as large as (or larger than) the size of the local state vector stored by each task these gates operate *consecutively on all* components. The compiler can build two streams prefetching the entire local state vector.

Since the memory access dominates the time needed to perform a quantum operation, it is crucial a) to minimize consecutive access to widely separated memory entries and b) to use a simple access

pattern allowing for efficient (compiler driven) prefetching. We investigate different ways to code the core routine, that determines the state vector components k, l and performs the computation on local qubits $q < N - p$.

version 1

```
do i=imin,imax,i1
   do k=i,i+i1n-1
      l=k+i1n
      ...
   enddo k
enddo i
```

version 2

```
do i=imin,imax,2
    i2=iand(i,i1n)
    k=i-i2+i2/i1n
    l=k+i1n
    ...
enddo i
```

Recoding of the core routine shifting from version 1 (nested loop) to version 2 (single loop) makes OpenMP loop parallelization simpler but destroys streaming because of additional "jumps" in the sequence of index k. To comprehend this we respectively present a part of a typical sequence of consecutive pairs of state vector elements (k, l) to be read from memory.

version 1	k	**0**	1	2	3	4	5	6	7	**16**	17	18	19	20	21	22	23	**32**
	l	**8**	9	10	11	12	13	14	15	**24**	25	26	27	28	29	30	31	**40**
version 2	k	**0**	2	4	6	**1**	3	5	7	**16**	18	20	22	**17**	19	21	23	**32**
	l	**8**	10	12	14	**9**	11	13	15	**24**	26	28	30	**25**	27	29	31	**40**

To halve the number of co-resident streams to be prefetched from memory we additionally split the computation of $|\psi_e\rangle$ and $|\psi_o\rangle$ into two sequential loops of version 1. This speeds up computation considerably. We measure that our fastest nested OpenMP parallelization of version 1 is about 25% faster than the dense coded version 2. This gain applies to the usage of 1,2,4 and 8 OpenMP threads.

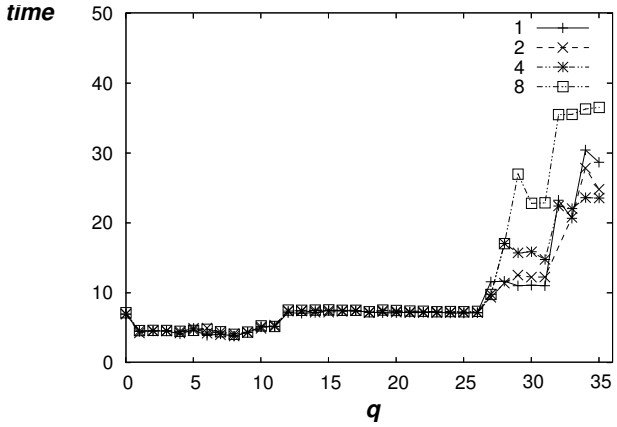


Figure 3. Timings for a Hadamard operation on qubit q at system size N=36 depending on the number of threads $T = 1, 2, 4, 8$ per MPI-task using $T * 2^p = 512$ processors.

Analyzing the time needed to compute H_q depending on the qubit q the operation acts on (see Fig.3), we can identify three regions according to different speeds of memory access. In case of $T = 1$ (equivalent to pure MPI-parallelization) we obtain fast computation for $q < N - p = 27$, because all state vector components involved are located within processor memory. Higher timings for $N-p \quad q < N-p+5 = 32$ arise from intra-node communication. The communication between

processors is mapped to shared memory access, which is slower than access to the memory associated to a single processor, but faster than MPI based inter-node communication for $q \quad 32$. Timings for parallelizations using 16 (32) threads per MPI process are not given in Fig.3, since computation gets slower by a factor of about 3 (6) compared to pure MPI. This is due to the machine architecture: each node is built up by 4 multichip modules each containing 8 processors. The memory access within a multichip module is faster than getting data from memory associated to another module.

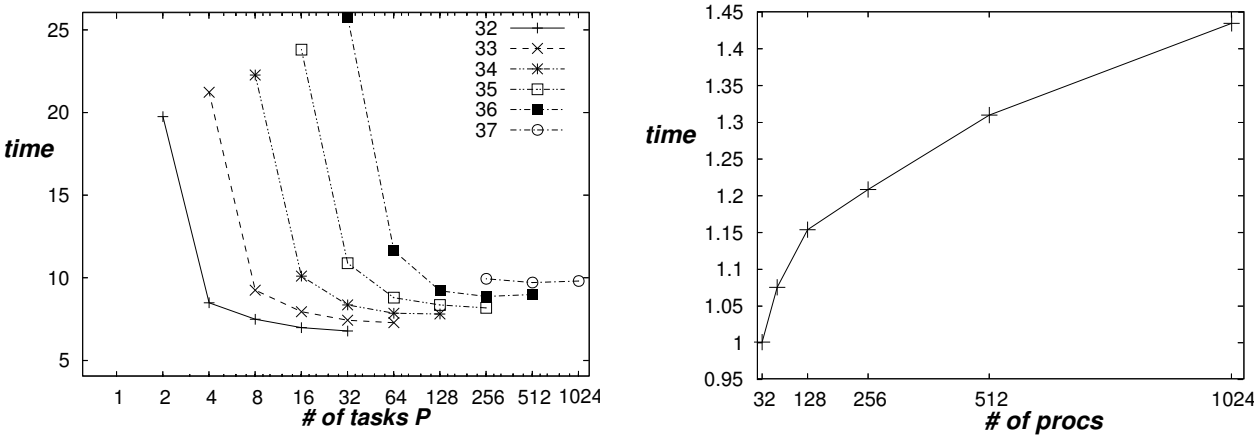


Figure 4. Left: average timings $t_{av}(N)$ for a Hadamard operation on different system sizes N depending on the number of MPI tasks $P = 2^p$ using $T = 2^t$ OpenMP threads with $t + p = N - 27$. Right: scaling of the minimal average timings for the system sizes $N = 32, 33, 34, 35, 36, 37$ using 32,64,128,256,512,1024 processors respectively.

Keeping the local size of the state vector fixed at 2 GB per processor we compare the time needed to compute Hadamard operations on all qubits for different choices for the number of tasks and threads. Fig.4 shows the average timings $t_{av}(N)$ for a complete Hadamard transformation $N \ ^1 \prod_{q=0}^{N} \ ^1 H_q$ on different system sizes N depending on the number of MPI tasks $P = 2^p$ and different numbers of OpenMP threads $T = 2^t$ respecting $t + p = N - 27$. Multiple measurements indicate a statistical timing error of max 5%. On this error level we identify the usage of one or two OpenMP-threads per MPI-task as optimal. Using 4 threads gives a slightly worse timing.

Taking the best average timing results normalized to $\min(t_{av}(32))$ from the l.h.s of Fig.4 for each system size N we observe the weak(=local size fixed) scaling behavior plotted on the r.h.s of Fig.4. Compared to the optimal (weak) scaling of a constant $\min(t_{av}(N))/\min(t_{av}(32)) = 1$ we still have an efficiency of 70% simulating a 37 qubit-system. Looking separately at the timings of H_q with $q < 32$ and $q \quad 32$ varying $N = 33, 34, 35, 36$ reveals that the timings follow the expectation of

$$\frac{t_{av}(N)}{t_{av}(32)} \approx 1 + \frac{(N-32)\, t_{\geq 32}}{32\, t_{<32}}$$

for $N = 33, 34, 35, 36$ within the 5% error level. Hence the efficiency loss simulating larger systems is due to the linearly increasing fraction of operations on nonlocal qubits $q \quad 32$ using internode MPI-communication. Observing approximately constant time $t(q, N)$ for a Hadamard operation on a fixed qubit $q \quad 32$ and varying N we can deduce, that our highly optimized parallelization of

the demanding memory bounded problem does not saturate the accumulated bandwidth on the IBM p690 up to the usage of 1024 processors an 3TB memory.

6. Two-Qubit Operations

A universal quantum computer also needs two-qubit operations such as the CNOT gate to incorporate qubit interaction. We illustrate the action of the $CNOT_{CT}$ gate on a two qubit state that flips the target qubit T if the control qubit C is set to $|1\rangle$

$$CNOT_{10} \begin{pmatrix} a_{00} \\ a_{01} \\ a_{10} \\ a_{11} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} a_{00} \\ a_{01} \\ a_{10} \\ a_{11} \end{pmatrix}. \tag{5}$$

The compute pattern of the CNOT-gate is very similar to the one given in Fig.1. The stride of the state vector components involved in the operation is given by the target qubit $|l - k| = 2^T$. Without presenting further details our simulator includes load balanced implementations of the controlled NOT and the controlled phase shift operations as fundamental two-qubit gates.

7. Results

An efficient parallelization technique was applied to the memory bounded problem of simulating ideal quantum computers, based on hybrid usage of MPI and inner node OpenMP communication using 1,2 and 4 threads. A compact state vector referencing reduces significantly cache misses produced by irregular access to widely separated parts of the memory. An algorithm built up from elementary one- and two-qubit gates scales very well on the IBM p690 up to the max. available memory using 1024 processors and 3 TB of memory (keeping the local state vector size fixed at 2 GB memory per processor).

8. Acknowledgments

Our presented investigations of simulating ideal quantum computers on high-end classical computer systems are part of an ongoing collaboration with H. De Raedt, K. Michielsen and K. De Raedt from the University of Groningen. We thank them for the initial incentive and a series of very instructive discussions.

References

[1] "Quantum Computation and Quantum Information", M.A. Nielsen and I.L. Chuang, Cambridge University Press, Cambridge, 2004.
[2] P. W. Shor, SIAM J. Computing 26, 1484 (1997).
[3] L. K. Grover, Phys. Rev. Lett. 79, 4709 (1997).
[4] L.M.K. Vandersypen, M. Steffen, G. Breyta, C.S. Yannoni, M.H. Sherwood, and I.L. Chuang, Nature 414, 883 (2001).
[5] D. P. Di Vincenzo, http://arxiv.org/abs/quant-ph/0002077.
[6] W. H. Zurek, Phys. Rev. D 24, 1516 (1981), Phys. Rev. D 26, 1862 (1982); E. Joos and H. D. Zeh, Z. Phys. B 59, 223 (1985).
[7] S. Lloyd, Science **261**, 1569 (1993).
[8] A. Barenco et al., Phys. Rev. A 52, 3457 (1995).

Cluster Computing

Adaptive Selection of Communication Methods to Optimize Collective MPI Operations

Olaf Hartmann[a], Matthias Kühnemann[a], Thomas Rauber[b], Gudula Rünger[a]

[a]Department of Computer Science, Chemnitz University of Technology, 09107 Chemnitz, Germany

[b]Department of Mathematics and Physics, Bayreuth University, 95445 Bayreuth, Germany

Abstract Many parallel applications from scientific computing use collective MPI communication operations to distribute or collect data. The execution time of collective MPI communication operations can be significantly reduced by a restructuring based on orthogonal processor structures or by using specific point-to-point algorithms based on virtual communication topologies. The performance improvement depends strongly on numerous factors, like the collective MPI communication operation, the specific group layout, the message size, the specific MPI library, and the architecture parameters of the parallel target platform. In this paper we describe an adaptive approach to determine and select a specific processor group layout or communication algorithm for the realization of collective communication operations with the objective of minimizing the communication overhead. In the case that a communication method is faster than the original implementation of the collective MPI communication operation, the specific communication method is applied to perform the communication operation.

1. Introduction

The execution of data or task parallel implementations of program applications can lead to scalability problems, especially for target platforms with a large number of processors [2]. To reduce the communication time of collective MPI communication operations different approaches can be considered [5,6]. One of them is the use of orthogonal processor groups performing two or more communication phases. An orthogonal processor group is obtained by arranging subsets of processors as two- or multi-dimensional grid. In [3] we have shown how the execution time of collective MPI communication operations can be significantly reduced by orthogonal processor groups for different target platforms and different MPI implementations. Performance improvements of 40% up to 70% can be obtained for LAM-MPI communication operations, like a `MPI_Bcast()` or `MPI_Allgather()`, on a Cray T3E-1200 and a Beowulf cluster.

Since only a specific collective MPI communication operation is replaced by two or more communication phases of orthogonal processor groups, the use of this approach is completely independent of the computation or communication structure of the parallel program application. This means that all applications using MPI operations for exchanging data can benefit from the improved communication method. The overhead caused by generating the processor groups is negligible and can easily be done by using a specific communicator handle for the corresponding row or column group.

However, it is not known in advance whether a performance gain can be achieved by an orthogonal processor layout and which of the numerous processor layouts may lead to an optimal performance gain. The reason is that the performance improvement depends on numerous factors, like the specific collective MPI communication operation, the message size, the total number of processors participating in the operation, the specific MPI library, and the architecture characteristics of the target machine [8,7]. In this paper we introduce an adaptive approach that determines and automatically selects the processor layout depending on the message size, such that the best performance improvement for a corresponding collective communication operation is achieved. In addition, we consider

specific realizations of collective communication using point-to-point operations. The fastest realization is then used in the parallel application program. The adaptive approach can be applied for each collective MPI communication operation and each parallel target machine providing MPI.

2. Design of Orthogonal Groups and Communication Algorithms

For a collective MPI communication operation on $p = p_1 \cdot p_2$ processors, an orthogonal group layout of processors can be arranged as row groups with p_1 processors and column groups with p_2 processors. The disjoint processor sets resulting from column groups are orthogonal to the row groups. Using the orthogonal processor group a collective MPI communication operation is performed in two phases. To perform more than two communication phases the rearrangement can be applied recursively to the row groups or the column groups, respectively. In [3] the generation and use of orthogonal processor groups is described in more detail.

Another possibility to reduce the communication overhead of a collective communication operation is an implementation based on a series of non-blocking point-to-point MPI communication operations. In this paper we consider a selection of the most promising algorithms. The algorithms are based on specific virtual communication topologies, where the exact algorithm depends on the specific collective MPI communication operation. These communication methods are integrated into a program library which provides a large variety of different realizations for each collective communication operation. Based on this library, a modeling tool can determine and select an optimal communication method depending on the message size for a given number of processors on a specific execution platform. We consider a star, a hypercube, a binomial tree and a ring as virtual communication topology to implement a corresponding collective MPI operation.

The user can perform the benchmark and the execution of the application program separately. This reduces the overhead caused by the performance benchmark when a large number of program starts of application programs is performed. The adaptive approach can easily be used by including a specific header file and linking the C program library to the program application.

3. Implementation of the adaptive approach

The basic idea of the approach is to execute a specific benchmark program that contains different implementations for each collective MPI operation with the objective of determining the fastest communication method for a specific interval of message sizes. As communication methods the vendor-specific collective MPI communication operation, orthogonal realizations with all possible two- and three-dimensional group layouts and two different communication algorithms based on virtual topologies are considered. Table 1 shows which communication topology is used for which collective MPI operations. For each collective MPI operation two algorithms P2P_A and P2P_B are implemented using non-blocking point-to-point (P2P) MPI communication operations. Benchmark programs of these algorithms are implemented as standalone C program library and can be used for each collective communication operation on each parallel target platform providing MPI. Based on the benchmark results, the fastest communication method is determined in an evaluation phase which has to be executed once for each combination of a target platform and an MPI implementation.

For each collective MPI communication operation the information gathered in the evaluation phase is stored in an information table, such that a specific interval of message sizes is mapped to the fastest communication method. This mapping can be used to select the fastest communication method in the execution phase in which the application program containing one or more collective MPI operations is executed.

MPI operation	P2P_A	P2P_B
`MPI_Bcast()`	binomial tree	binomial tree (scatter) + hypercube (allgather)
`Gather()`	star	binomial tree
`Scatter()`	star	binomial tree
`Reduce()`	star	binomial tree
`Allreduce()`	binomial tree (reduce) + binomial tree (bcast)	hypercube
`Allgather()`	ring	hypercube

Table 1: Two communication algorithms `P2P_A` and `P2P_B` are implemented to perform a collective MPI communication operation. The algorithms use non-blocking point-to-point MPI operation.

4. Parallel Target Platforms and Experimental Results

This section gives an overview of the hardware characteristics of the parallel target platforms that are used to investigate and evaluate the performance behavior of collective MPI operations with and without optimized communication methods. Section 4.2 gives a summary of the performance results on these platforms achieved by optimized communication method in isolation.

4.1. Target Platforms

The runtime experiments are performed on a Beowulf cluster (CLiC), a dual Xeon cluster, a Cray T3E and a IBM Regatta p690+ cluster, which have the following characteristics.

- The Beowulf Cluster `CLiC` (`C`hemnitzer `Li`nux `C`luster') is built up of 528 Pentium III processors clocked at 800 MHz. The processors are connected by a fast-Ethernet network which can be used by LAM MPI 6.5.6 and MPICH 1.2.5.2.
- The `Dual Xeon Cluster` is built up of 16 nodes consisting of two Xeon processors clocked at 2 GHz each. The nodes are connected by a fast-Ethernet network and a high performance interconnection network that uses Dolphin SCI interface cards. The SCI network is connected as two-dimensional torus and can be used by the ScaMPI (SCALI MPI) library [1]. The fast-Ethernet based network is connected by a switch and can be used by two portable MPI libraries, LAM MPI 6.5.6 and MPICH 1.2.5.2.
- The `Cray T3E-1200` uses a bidirectional three-dimensional torus network to connect the nodes each containing a DEC Alpha 21164 processor with 600 MHz. The six communication links of each node are able to simultaneously support hardware transfer rates of 600 MB/s.
- The `JUMP` (`JU`elich `M`ulti `P`rocessor) is a IBM Regatta p690+ cluster, that is built up of 41 SMP nodes; each node consists of 32 Power4+ processors clocked at 1.7 GHz. The nodes are interconnected by a High Performance Switch (HPS). As Message-Passing library a proprietary implementation of MPI is used for all runtime experiments, that is part of the Parallel Environment v4.1 developed by IBM.

4.2. Experimental results in isolation

A specific set of different orthogonal group layouts depends on the total number of available processors of a parallel target platform. In general, we arrange the processor groups into all possible two-dimensional or three-dimensional grid layouts based on the total number of processors participating in the communication operation. On the T3E and the CLiC, 96 processors have been used for the experimental evaluation leading to 10 different two-dimensional group layouts ($2 \times 48, 3 \times 32, 4 \times 24, 6 \times 16, 8 \times 12, 12 \times 8, 16 \times 6, 24 \times 4, 32 \times 3, 48 \times 2$). Additional performance tests with a total number of 48 processors allow 8 different two-dimensional group layouts

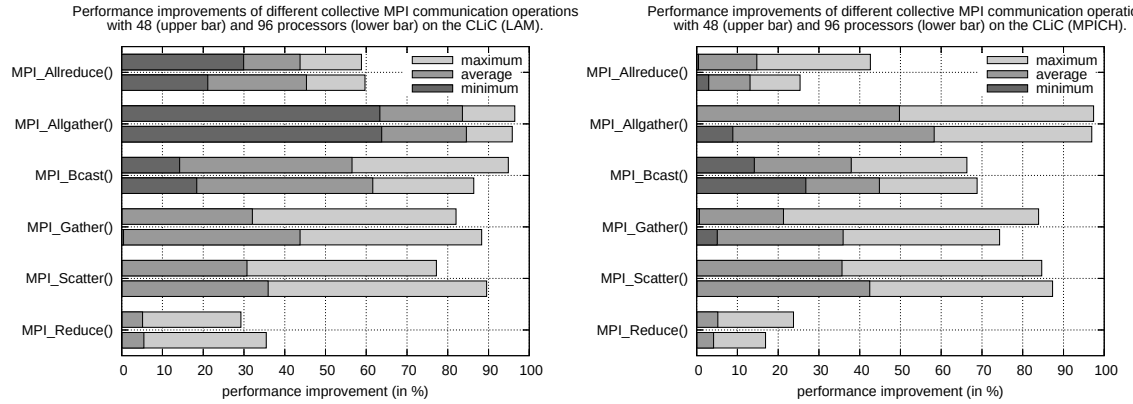


Figure 1. Summary of the performance improvements achieved by different communication methods for collective MPI operations. The diagrams show the improvements obtained with LAM-MPI (left) and with MPICH (right) on the Beowulf cluster CLiC. The bars represent the minimum, average, and maximum performance improvements over all message sizes for a total number of 48 processor (upper bar) and 96 processors (lower bar) for each MPI operation.

$(2 \times 24, 3 \times 16, 4 \times 12, 6 \times 8, 8 \times 6, 12 \times 4, 16 \times 3, 24 \times 2)$ on both platforms. On the JUMP, runtime tests with 64 processors are considered; 32 processors are available on the dual Xeon cluster. The message sizes for the runtime tests are between 1 KByte and 1024 KByte.

4.2.1. Beowulf cluster CLiC

The standard implementations of MPICH and LAM-MPI have the drawback that an unsuitable use of communication protocols may lead to significant performance degradations on several execution platforms. This can be observed, e.g., for the original `MPI_Gather()`, `MPI_Scatter()` and `MPI_Allgather()` operation on the CLiC. Different communication protocols, like the eager and the rendezvous protocol, are used to optimize the data throughput for smaller and larger message sizes. The performance degradations are caused by an unsuited selection of the specific message size at which the system switches to a different communication protocol. This disadvantage can be compensated using orthogonal processor groups. The reason is that an orthogonal realization use the significantly faster rendezvous protocol for larger messages in the second communication phase, in comparison to the slower eager protocol for smaller messages used by the original MPI operation.

We have measured the performance improvements achieved by the optimized communication methods for all message sizes considered. Figure 1 shows the minimum, average, and maximum performance improvements obtained. The figure depicts the improvements for a total number of 48 and 96 processors on the CLiC using LAM-MPI (left) and MPICH (right). Since the adaptive approach determines the most efficient communication method for separate intervals of message sizes, the figures give a good survey of the expected reduction of the communication overhead in the context of parallel program applications on the CLiC. Finally we notice that for an `MPI_Bcast()` and an `MPI_Allgather()` operation, a point-to-point based algorithm usually leads to the best performance enhancements in contrast to the `MPI_Gather()`, `MPI_Scatter()` and `MPI_Reduce()` operation, where an orthogonal realization often lead to the best improvements.

4.2.2. Dual Xeon Cluster

The network interconnection based on the Fast-Ethernet standard can be used by the LAM-MPI and the MPICH implementation. Since the same portable MPI-library and network system are in-

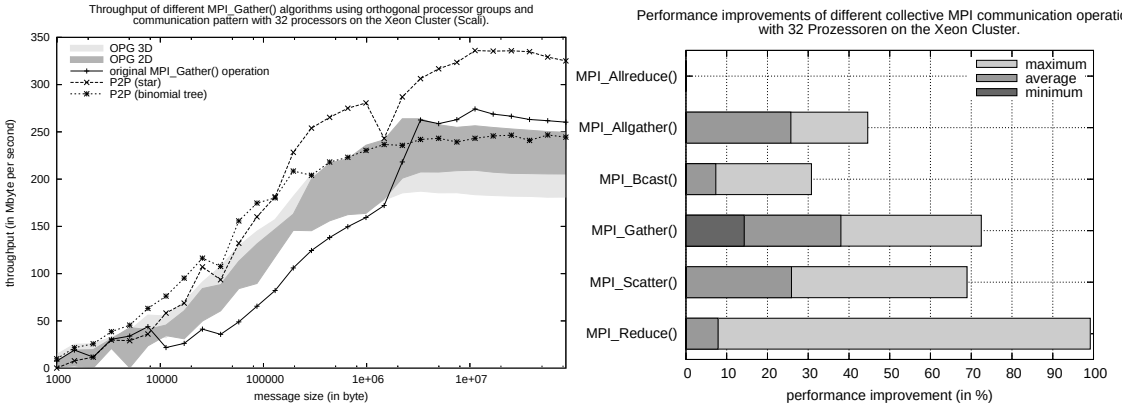


Figure 2. The left figure shows the throughput of different `MPI_Gather()` communication methods with 32 processors on the dual Xeon cluster using the SCI network. The grey areas depict the throughput achieved by orthogonal processor groups representing the fastest and the slowest group layout with two communication phases (orthogonal processor groups `OPG 2D`) and three communication phases (orthogonal processor groups `OPG 3D`). The right figure shows a summary of the performance improvements achieved by optimized communication methods for all collective MPI communication operations on the dual Xeon cluster with ScaMPI/SCI for 32 processors.

vestigated for the same collective MPI communication operations, similar experimental results are obtained as on the CLiC. The performance improvements achieved by the optimized communication methods are also significant for most of the collective MPI operations corresponding to the smaller total number of 32 processors.

In spite of the fact that the collective communication operations in ScaMPI are optimized for the SCI network architecture, a clear performance improvements can be obtained for all collective MPI operations using different communication methods, see Figure 2 (right) for all collective MPI operations. Figure 2 (left) shows the throughout of different implementations of the `MPI_Gather()` operation with ScaMPI/SCI in detail.

4.2.3. Cray T3E-1200

An optimized proprietary message-passing library is provided on the Cray T3E-1200 to use collective MPI communication operations. But nevertheless the orthogonal realizations show significant performance improvements for various collective operations on this platform. Except for the single-accumulation operation `MPI_Reduce()` and the multi-accumulation operation `MPI_Allreduce()` consistent improvements are obtained for all other collective MPI operations. As examples, Figure 3 shows the data throughput for different implementations to perform a single-broadcast operation (left) and a scatter operation (right). Both point-to-point communication algorithms lead to a performance gain for the `MPI_Bcast()` operation. Figure 4 (left) shows a summary of the performance improvements for all collective MPI operations on the T3E.

4.2.4. JUMP Cluster

Similar to the T3E, a proprietary message-passing library developed by IBM is provided as part of the Parallel Environment V 4.1 to perform a collective MPI communication operation on the JUMP. But also for this implementation all collective MPI communication operations show performance improvements in the average using orthogonal realizations and point-to-point algorithms over a wide range of message sizes for a total number of 32 and 64 processors, see Figure 4 (right).

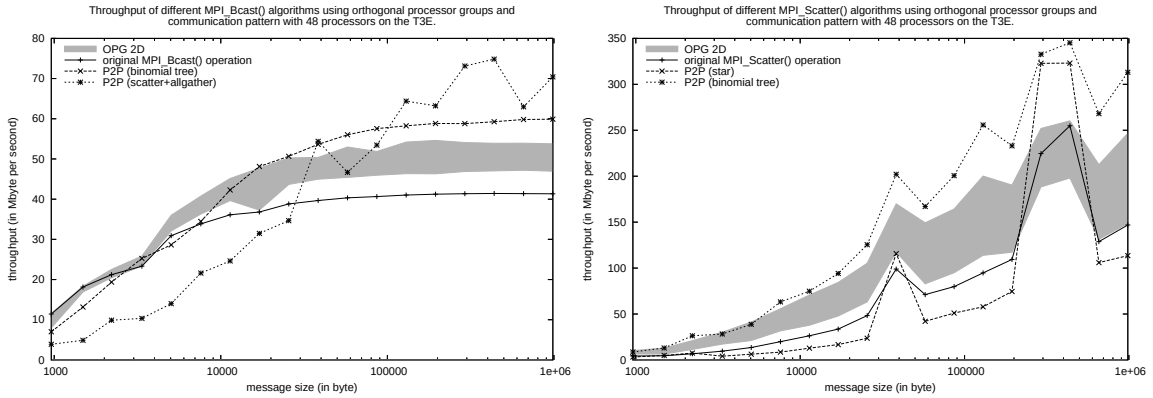


Figure 3. Throughput of different `MPI_Bcast()` (left) and `MPI_Scatter()` communication methods (right) with 48 processors on the Cray T3E-1200.

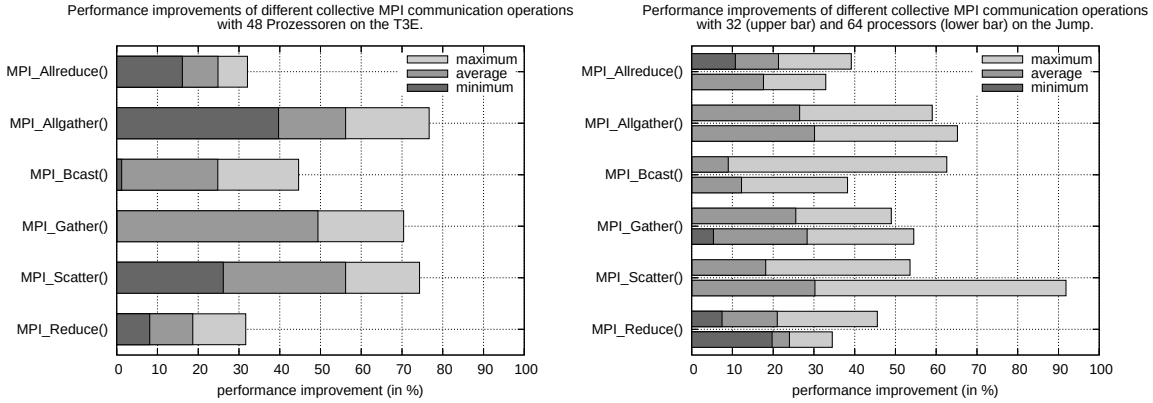


Figure 4. Summary of the performance improvements achieved by optimized communication methods for collective MPI communication operations. The figures show the minimum, average and maximum improvements over all message sizes obtained on the Cray T3E-1200 (left) for a total number of 48 processors and on the IBM Regatta cluster (right) for a total number of 32 processors (upper bar) and 64 processors (lower bar).

5. Runtime Tests of Parallel Program Applications

We consider the optimized communication methods in the context of complex program applications in order to verify the performance improvements achieved in isolation. For this purpose we apply the adaptive approach to reduce the communication overhead of a Jacobi iteration and a solution method of ordinary differential equations on the target platforms described in Section 4.1. Section 5.1 presents the runtimes of different program implementations of the Jacobi iteration and Section 5.2 considers the parallel Adams methods PAB and PABM.

5.1. Parallel Jacobi Iteration

We consider three different ways to implement the Jacobi iteration in a data parallel way based on a row-wise and a column-wise distribution of the matrix A. For both distributions the computational work for computing the new entries of the next iteration vector $x^{(k)}$ is the same and is equally allocated to the processors. For systems of size n each processor performs $\lceil \frac{n}{p} \rceil \times n$ multiplications

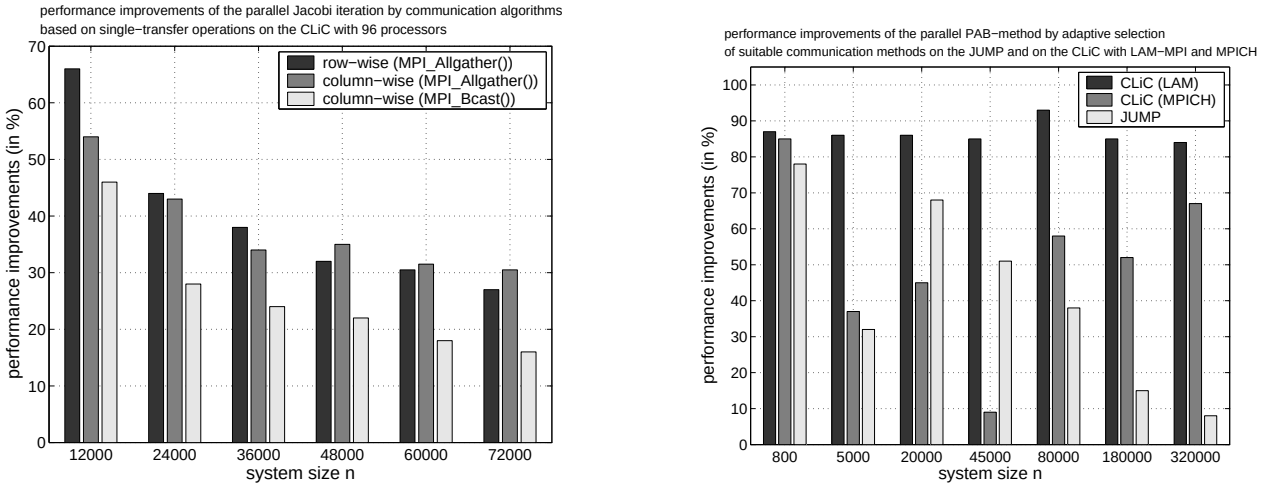


Figure 5. The left diagram shows runtime improvements for different implementations of the parallel Jacobi iteration achieved by optimized communication methods for different system sizes on the CLiC (LAM-MPI) for 96 processors. The right figure shows performance improvements of the parallel PAB Method achieved by an adaptive selection of suitable communication methods of the `MPI_Allgatherv()` operation on the CLiC (LAM and MPICH) for 96 processors. Additional performance improvements are depicted for identical system sizes on the JUMP for 64 processors.

and about the same number of additions in each iteration. But because each processor computes different parts and each processor needs the entire new iteration vector $x^{(k)}$ in the next iteration step, different communication operations are required for the implementations.

The row-wise realization uses a `MPI_Allgather()` operation to distribute the intermediate result of vector $x^{(k)}$ to all processors participating in the computation. For the column-wise realization, either a `MPI_Allreduce()` and `MPI_Allgather()` operation or a `MPI_Reduce()` and `MPI_Bcast()` operation can be used to distribute the intermediate result of vector x. Figure 5 (left) shows the performance improvements of different implementations of the parallel Jacobi iteration using the adaptive approach to select a suitable communication method on the CLiC (LAM-MPI) for 96 processors. In most cases, the adaptive approach selects point-to-point algorithms for performing the collective communication operations, since these algorithms lead the best performance improvements. On the target platforms considered, the point-to-point algorithms are slightly faster than the orthogonal realizations for the most system sizes.

5.2. Parallel Adams-Bashford Methods

Parallel Adams methods are variants of general linear methods for solving ordinary differential equations (ODEs) $\mathbf{y}'(t) = \mathbf{f}(t, \mathbf{y}(t))$ proposed in [4]. The name was chosen due to a similarity of the stage equations with classical Adams formulas. General linear methods compute several stage values $\mathbf{y}_{\kappa,i}$ in each time step κ which correspond to numerical approximations of $\mathbf{y}_{\kappa,i} = \mathbf{y}(t_\kappa + a_i h)$ with abscissa vector (a_i), $i = 1, ..., K$, and stepsize $h = t_\kappa - t_{\kappa+1}$. The stage values of one time step are combined in the vector $\mathbf{Y}_\kappa = (\mathbf{y}_{\kappa,1}, ..., \mathbf{y}_{\kappa,K})$. For an ODE system of size n, this vector has size $n \cdot K$.

A `MPI_Allgatherv()` operation is used to distribute the intermediate result. Figure 5 (right) shows the average performance improvements that are obtained by an adaptive selection of communication methods to perform the `MPI_Allgatherv()` operation on the CLiC and the JUMP.

On the CLiC the runtime results are investigated for both portable MPI implementations LAM and MPICH for a total number of 96 processors and different system sizes. Using LAM-MPI a performance gain of up to 90% can be observed. Based on the MPICH implementation, also a significant reduction of the communication overhead is obtained using optimized communication methods. Furthermore, the good performance results could be confirmed also for a propietary MPI implementation on the JUMP cluster for 64 processors, see also Figure 5 (right). Again, the point-to-point realizations are selected by the adaptive approach, since these are about 10% faster than the orthogonal realizations for the system sizes considered.

6. Conclusions

In this paper we have considered an adaptive approach to select suitable orthogonal group layouts and point-to-point communication algorithms with the objective of reducing the execution time of collective MPI communication operations. An adaptive selection is crucial in reducing the communication overhead, since the resulting performance improvements depend on numerous factors. The resulting overhead in the evaluation phase of the adaptive approach is small compared to the possible performance gain. Especially for implementations of collective operations of portable MPI libraries, like LAM or MPICH, significant improvements of the communication time are obtained, since these operations are not optimized for the underlying hardware architecture of the target platforms. But also for optimized collective communication operations of proprietary MPI libraries significant performance improvements are achieved by an adaptive approach. The experimental results show that the adaptive selection of different communication methods depending on the message size leads to average performance improvements for most of the collective MPI operations, since at least one of the optimized communication methods is almost always faster than the vendor implementation.

References

[1] Scali / ScaMPI commercial MPI on SCI implemetation. http://www.scali.com/.

[2] A. Hoisie, O. Lubeck, and H. Wasserman. Performance and Scalability Analysis of Teraflop-Scale Parallel Architectures Using Multidimensional Wavefront Applications. *International Journal of High Performance Computing Applications*, 14(4):330-346, Sage Publications, 2000.

[3] M. Kühnemann, T. Rauber, and G. Rünger. Optimizing MPI Collective Communication by Orthogonal Structures. In *Cluster Computing - The Journ. of Networks, Software Tools and Applications, Special Issue on Cluster Computing in Science and Engineering*, 8(4), Kluwer, 2005.

[4] P.J. van der Houwen and E. Mesina. Parallel adams methods. *J. of Comp. and App. Mathematics*, 101:153–165, Elsevier, 1999.

[5] E.W. Chan, M.F. Heimlich, A. Purkayastha, R.A. van de Geijn. On Optimizing Collective Communication. In *Proc. of Int. Conference on Cluster Computing*, IEEE, Sep. 2004.

[6] G.D. Benson, Cho-Wai Chu, Q. Huang, S.G. Caglar. A comparison of MPICH Allgather Algorithms on Switched Networks. In *Lecture Notes in Computer Science 2840*, Springer, Sep. 2003.

[7] S.S. Vadhiyar, G.E. Fagg, J. Dongarra. Automatically tuned collective communications. In *Proc. of SC99: High Performance Networking and Computing*, ACM/IEEE, Nov. 1999.

[8] S.S. Vadhiyar, G.E. Fagg, J. Dongarra. Performance Modeling of Self Adapting Collective Communications for MPI. *Los Alamos Computer Science Institute Symposium*, 2001.

Fault-Tolerant Master-Worker over a Multi-cluster Architecture *

Josemar Souza[abc], Eduardo Argollo[a], Angelo Duarte[ab], Dolores Rexachs[a], Emilio Luque[a]

[a] CAOS - Computer Architecture and Operating Systems Department, Universitat Autònoma de Barcelona (UAB), 08193, Barcelona, Spain

[b] CEBACAD - High Performance Computation Center of Bahia, Universidade Catolica do Salvador (UCSAL), Av. Cardeal da Silva, 205, 40.220-140, Salvador, Bahia, Brazil

[c] Universidade do Estado da Bahia (UNEB), Rua Silveira Martins, 2555, 41.150-000l, Salvador, Bahia, Brazil

Abstract: The growth of clusters into cluster collections increases potential points of failures, requiring the implementation of a fault-tolerance scheme. The CoHNOW is organized as a hierarchical master-worker scheme and clusters may be geographically distributed and interconnected by Internet. This paper describes a system of Fault-Tolerant protection by Data Replication (FT-DR), based on preserving critical functions by on-line dynamic data replication. The system-model target is to detect failures in any of the system functional elements and to tolerate this failure by recovering system consistency, guaranteeing the completion of the work in progress (recovery procedure). The model is designed to tolerate more than one simultaneous failure. There are three distinct phases for model-fault tolerance activities: startup, normal execution including failure detection monitoring, and failure recovery. The system is oriented for general master-worker applications running on CoHNOW and is transparent both for user and application. The master-worker environment requirements to support all these capabilities and the runtime overhead are under evaluation.

1. Introduction

Joining geographically distributed dedicated heterogeneous networks of workstations (HNOW) through Internet can be an inexpensive approach to achieving data-intensive computation. A collection of HNOW (CoHNOW) can be efficiently organized for parallel-application execution as a hierarchical master/worker scheme where each cluster is a master/worker itself [5]. This kind of CoHNOW presents several potential points of failure, namely, each cluster computer, local area networks and Internet network interconnection. The lack of quality guarantees for any of these components and the possibility of wasting considerable computational effort requires the implementation of a fault-tolerance scheme.

The general approach to protecting distributed parallel applications is to implement some sort of checkpoint strategy [4]. Currently, certain research groups are investigating the Fault-Tolerance problem in parallel applications with distributed memory.

We have developed a model that provides functional fault tolerance for all clusters' hosts by means of data replication. The model's target is to detect failures in any of the system functional elements and to tolerate this failure by recovering system consistency, guaranteeing the completion of the work (recovery procedure). In our model, if more than one failure occurs during the system-recovery procedure, these failures are considered to be concurrent. The model can be configured to admit the possibility of tolerating a number "C" of concurrent failures.

*The first author is grateful to the Universitat Autònoma de Barcelona (UAB), the Universidade Catolica do Salvador (UCSAL) and Universidade do Estado da Bahia (UNEB), for academic and financial support given during this work.

The proposed system of fault-tolerant protection by data replication (FT-DR), based on preserving critical functions by on-line dynamic data replication, is oriented for general Master/Worker (MW) applications running on CoHNOW and is user and application transparent. The model corresponds to a middleware transparent to the user, implemented as a layer between the application and the library on the message passing MPI.

For our architecture, based on the hierarchical Master/Worker (MW) model (all workers run the same program and there is no need to checkpoint [3] workers), a data-replication approach should be less computing expensive (low overhead) because it avoids the checkpoint cost of replicating all program states (memory, opened files, and code itself) for all machines, as was proved by Weissman in [6].

Weissman does not carry out data replication; he undertakes replicas of complete function/task. Our approach is similar, because protective workers act as ”sleeping masters” that will be enabled to receive the copies of the data and ”then go back to sleep”. When a master fault is detected, a ”sleeping master” ”awakes” and is converted into an enabled master.

Weissman [6] explores two options: checkpoint-recovery, quantitatively, and wide-area replication, as the means of achieving fault tolerance. His experimental results suggest that checkpoint-recovery can be preferable for small problems, whilst replication is preferable for larger problems, i.e. precisely those that use WAN environments (as in our case). The results also show that it is possible to predict overheads for the two methods.

The master-worker environment requirements needed to support all the proposed model capabilities and the runtime overhead are under evaluation.

The following sections present the study in further detail. Section 2 describes the system architecture. Our Functional Model is explained in Section 3. In Section 4, the experiment results are shown, and finally, conclusions and further work are presented in Section 5.

2. System Architecture

To develop the Functional Model of Fault-Tolerance Cluster Computers based on wide-area data replication (FT-DR), this was conceived as a Master/Worker (MW) hierarchical architecture, as shown in Figure 1.

The hierarchical CoHNOW architecture presents four main logical functionalities: the main master (MMT) is the master (MT) in which the computation is started and where initially all data to be processed resides. The sub-cluster (SC) represents a remote cluster. The sub-master (SMT) computation is a subset of the main-master (MMT) problem and receives the data through the wide-area network link. To isolate the local network from the SC traffic and to provide a reliable transparent communication between MMT and SMT, the architecture presents the communication manager function (CM). Finally, the computational data is processed by the workers (WK) in all clusters. An SC is seen by MMT as a worker with great latency of communication and ”great” computation power.

The MW paradigm was chosen because it allows portability to a wide range of applications. In this architecture, any of the hosts of a CoHNOW can be a Main Master (MMT), Worker (WK), Communication Manager (CM), and Sub Master (SMT). In other words, any host of the CoHNOW can take over other functions and data of the CoHNOW in case of failure. However, cluster heterogeneity should be borne in mind when choosing the function of the hosts. In this architecture (hierarchical MW), for each Sub Cluster (SC, also called remote clusters), there will be a CM in the Main Cluster (MC, also called local HNOW).

The CM is responsible for maintaining a local MPI connection and a Socket connection with the

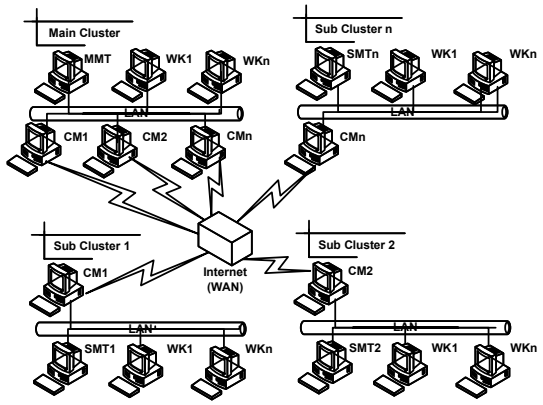


Figure 1. System Architecture

remote HNOW, therefore acting as a gateway for all incoming and outgoing data. The CM function can be a dedicated machine or a logical machine (a process running on a host of the cluster). All CoHNOW hosts have two network adapters. One of these network adapters is used to access the local network (NET) and the other is used to access the remote network (Internet). Access to the remote network (Internet) by one of the cluster hosts is given when that host takes over CM functions, in the case of failure in one CM [1].

This architecture is enhanced for fault-tolerant support with the usage of a developed fault-tolerant communication library by the CM functionalities. In order to support CM failures (transitory) and because the CM workstation needs two network cards (intra-cluster and inter-cluster), this two-network card facility is included in all workstations so that any computer can assume any functionality.

3. Functional model

In our model we use wide-area data replication, which is an important technique for increasing computer system availability of this. This occurs when a data set is copied and attributed to more than one unit of processing. This Fault-Tolerance technique (application) has an associated cost: wide-area data replication consumes computational and input/output resources, decreasing the performance of the system as a whole. In our model, Fault Tolerance does not imply physical redundancy; we are interested in ”functional redundancy”, in other words the functions and the data are taken over by other hosts of the cluster in case of failure in any component of the CoHNOW.

FT-DR provides Fault Functional Tolerance for all the elements of a cluster by means of Data Replication. The goal of the model is to detect faults in any of the functional elements within the system and to allow the fault to recover so as to ensure system consistency and to guarantee the conclusion of the work, despite performance loss. In our model, if more than one fault occurs during the system’s recovery procedure, these faults are considered as concurrent. The proposed model allows selecting the number of concurrent faults (”C”) - protection level - to be tolerated by the system. The model corresponds to a middleware transparent to the user. This is a layer between application and the library on the message passing MPI. (Figure 2).

In our model, the protected elements are: MMT, WK, CM, SMT and Network (NET). All failures in the CoHNOW hosts are considered permanent. Failure of the network (local or remote) is considered transitory, in other words, a failure can occur. However, we presuppose that the network will

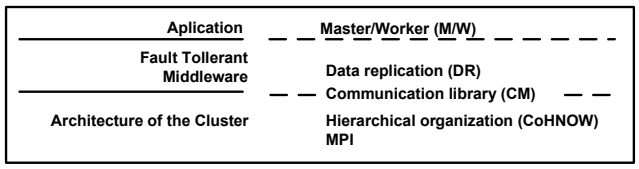


Figure 2. Layer model

always return to its activity [1]. Faults in the communication between MC and SC are considered transitory. The ”worker” components are not individually replicated-protected because their ”pool” structure allows implementing a task re-allocation in case of a worker fault. Failures of the networks are always considered as ”non-permanent” faults.

To protect the critical component (none-replicated) such as masters and cluster CM, the data and functions of these elements are replicated in certain workers. ”Replicas” of the critical components include the updated application, ”data structures” and their specific codes. These replicas are ”sleeping” and they are temporarily ”woken-up” for the updating of ”data” and ”control” values. Replicas are hosted in certain workers (background function) identified as protective workers (PWK).

The master information includes application data (problem and result data) and configuration data (a list of active workers and the list of assigned and remaining tasks).

The detection of an occurred failure is carried out through verifying the status of the cluster nodes. If any node of a cluster is inactive for a certain time (heartbeat), it is considered a failure node; the recuperation of this failure is then activated. In other words, ”inactivity” implies that there is a process monitoring (explicit or not) which can inform about the ”state” of the resource (machine or network).

The model is based on architecture having several workers with the same functionality and different data (Figure 3). The master has a copy of the data, and in the case of worker failure the running task will be reassigned to a different worker. The master has data from workers and monitors (watchdogs) the workers to detect their faults and to reallocate work. We consider the task of a worker in a data set as an atomic operation that works under a transactional scheme. If a task is allocated to

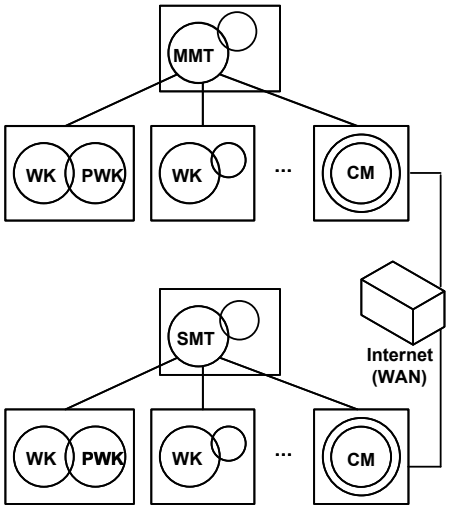


Figure 3. Logical functionalities

From	Action	To
MT	Replicate	PWKn/WKn and WKn
MT	Reconfigure	PWKn/WKn and WKn
MT	Protect	PWKn/WKn and WKn
MT	Protect	CM
MT	Reconfigure	CM
PWKn/WKn	Protect	MT
CM	Protect	SC

Figure 4. Protection scheme

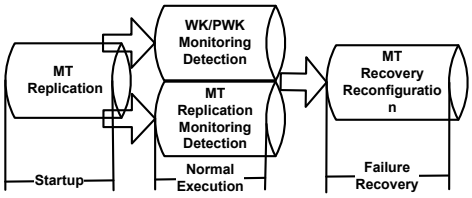


Figure 5. Fault tolerance activities

a worker, it runs and returns its results; another task can then be allocated to this worker. If it fails, the task will be allocated to other workers. MMT and SMT play the same role and use the same protection mechanism.

The MMT and SMT are responsible for dividing work into smaller tasks and for distributing tasks in an on-demand scheme to each worker. To solve the computational problem, we consider that the master communicates with the worker only to send a task and receive the results and that there is no communication between workers (transactional model). Our work considers a master-worker model where the master contains all initial application data and these data are divided into a certain amount of smaller data tasks, which in turn are also replicated.

The fault-tolerant scheme is summarized in Figure 4, where we show who is responsible (From) each one of the activities (Action), as well as the destination (To) of this activity.

There are three different phases for fault-tolerance activities in our model: startup, normal execution (replication, monitoring and detection (”protect”)), and failure recovery (reconfiguration) (Figure 5). Startup is the action of ”setup” for the cluster and in order to replicate the initial data. Normal execution represents prevention, including monitoring, fault detection and replication (protect). Fault recovery represents the action of recovering (i.e., so as to reconfigure) the cluster in the event of some of the foreseen faults in our model.

3.1. System Operation

After users define the level of protection, the three phases continue with their functions (Figure 5):

a) At startup time, the master information is replicated to certain workers. These special workers, called ”protective” workers (PWK), are those that can assume the master role in the event of master failure.

b) At normal execution without failures, before sending a task to a worker, the master multicasts this task index to all protective workers and to the target worker so that the PWK can maintain the consistency of master information. After completion of a task, each worker multicasts the result to the master and to each PWK. The PWK temporarily activates the ”sleeping master” to receive the ”worker-task results” and update the control information about the workers’ tasks. Therefore, the

master information is always updated in all ”C” PWK.

For detecting failures, certain messages are added to the original master-worker scheme: whenever a worker multicasts a result, the master and PWK send back a result-receipt acknowledgement to the worker; from time to time, each worker sends a heartbeat message to the master indicating that he is alive; from time to time the master also multicasts a heartbeat message to the PWK.

With these actions, it is possible to detect system failures. A master can detect any worker failure by counting the missing consecutive heartbeats for this worker. Whenever this number reaches a given limit, the worker is assumed to have failed. Workers can detect a master or PWK failure by not receiving their result acknowledgements, and PWK can also detect a master failure by not receiving a certain number of consecutive heartbeats.

c) In the event of any failure being detected, the system enters into the recovery phase (failure recovery). If failure in a worker is detected by the master, the master ascertains whether this worker is a basic WK, a CM or a PWK. A basic worker is simply dismissed from the system and all PWK are advised to modify their master information copy. The faulty worker's lost task will be reassigned to another worker (MT defines which worker, based on a priorities table). If the failed worker is a communication manager, the master not only acts by isolating the worker and updating the PWK state but also selects another worker to assume the CM function.

To solve a PWK failure, the master selects a basic worker to assume PWK functionality and sends all its information to this worker. It is of importance to observe that, while PWK failure is not recovered, there will be no work distribution and the workers will naturally be stopped, waiting for the failed PWK acknowledgement. The new PWK sends a special acknowledgement to the workers, and the workers update their multicast table.

Whenever a master failure is detected by one of the PWK, the PWK start a protocol for choosing the new master. This new master acknowledges the workers with this new information.

Figure 6 is a flow diagram showing a simplified model of the protection and recovery procedure interactions.

4. Validation

To validate and evaluate our proposal, we implement FT-DR, using as the test/benchmark program an MW version of the matrix multiplication (MM) algorithm [2] [5]. To evaluate the efficiency of the solution, experiments using an algorithm without FT-DR-and the same algorithm when modified-were performed, including FT-DR routines.

4.1. Results

The results of the experiment were obtained by using two geographically distributed clusters: one located at the Universidade Catlica do Salvador (UCSAL), Salvador, Bahia, Brazil and another located in the Universitat Autnoma de Barcelona (UAB), Barcelona, Spain,a heterogeneous environment. As a library for message passing, we use MPICH 1.2.5; an implementation supported in the standard MPI 1.1. The algorithms used in these experiments were compiled (gcc 3.2.3) and run on Kernel 2.4.20, the GNU/Linux Operating system. The LAN used in each one of the clusters is an Ethernet standard of 10 Mbps. In addition to UCSAL (worse condition), the connection with the Internet (WAN) was made through a non-dedicated link of 512 Kbps.

The results were obtained through this experiment (Figure 7) by means of executing the MM algorithm without FT-DR and the same algorithm when modified, including FT-DR routines (WK protection), at different time and days.

As the obtained data may show slight variation (order matrix multiplication of 1000, 1500 and

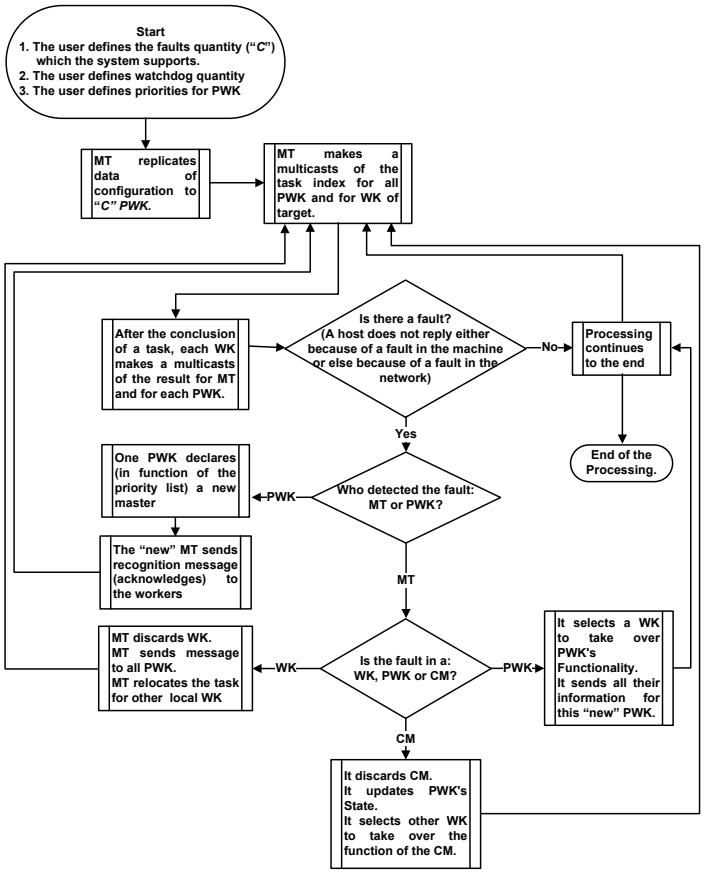


Figure 6. System operation

Matrix order	1000	1500	2000
Overhead	5,95%	2,02%	0,69%

Table 1
Overhead (%) after including the FTDR routines

2000 were used), we bear in mind that throughput and latency of connection is variable and influenced by the conditions of the network at the time of the experiments. As it is shown in Table 1, the insertion of FTDR routines (WK protection) in the MM algorithm, introduces an overhead (Execution Time with FTDR - Execution Time without FTDR) that decrease with high orders matrices.

5. Conclusions and further work

We have developed a dynamic replication scheme (FT-DR) for fault tolerance in wide-area clusters, for a common parallel programming paradigm, the Master/Worker.

Our FT-DR model in the preliminary experiments shows a rear-constant overhead and confirms the observations of Weissman [6] that replication-data model for a workstation environment is an excellent option for inserting Fault Tolerance into algorithms that solve great computational problems executed in CoHNOW environments.

Future work includes an analysis and evaluation of varying design options, in order to offer the

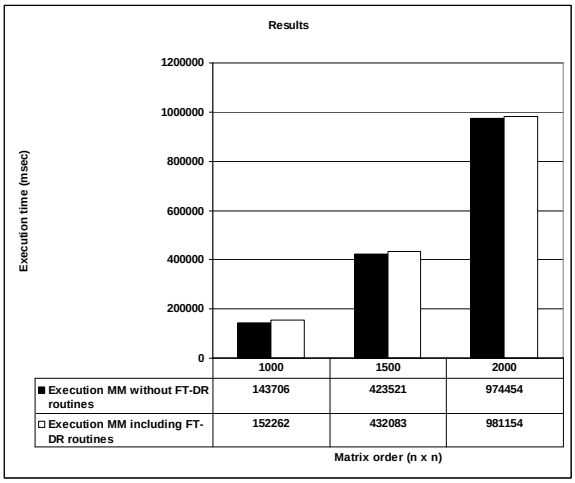


Figure 7. Execution Time and Overhead without and with FTDR routines

user different possibilities as regards both the application and user requirements.

Experimentation with additional MW applications is required. Future work will also include experimentation-critical elements, which our model (FT-DR) currently allows to fail.

References

[1] E. Argollo, J. R. Souza, D. Rexachs, and E. Luque. Efficient execution on long-distance geographically distributed dedicated clusters. *Lecture Notes in Computer Science*, v.3241:p.311–318, 2004.

[2] Olivier Beaumont, Vincent Boudet, Fabrice Rastello, and Yves Robert. Matrix multiplication on heterogeneous platforms. *IEEE Transactions on Parallel and Distributed Systems*, v12, n.10:p.1033–1051, 2001.

[3] George Bosilca, Aurelien Bouteiller, Franck Cappello, Samir Djilali, Gilles Fedak, Cecile Germain, Thomas Herault, Pierre Lemarinier, Oleg Lodygensky, Frederic Magniette, Vincent Neri, and Anton Selikhov. Mpich-v: Toward a scalable fault tolerant mpi for volatile nodes. *SC '02: Proceedings of the Proceedings of the IEEE/ACM SC2002 Conference*, page p.29, 2002.

[4] E. N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys, ACM Press*, v.34:p.375–408, 2002.

[5] A. Furtado, A. Rebouas, J. R. Souza, D. Rexachs, and E. Luque. Architectures for an efficient application execution in a collection of hnows. *Lecture Notes in Computer Science*, v.2474:p.450–460, 2002.

[6] Jon B. Weissman. Fault tolerant wide-area parallel computing. *Lecture Notes in Computer Science*, v.1800:p.1214–1225, 2000.

A distributed scheme for fault-tolerance in large Clusters of Workstations *†

Angelo Duarte[ab], Dolores Rexachs[a], Emilio Luque[a]

[a]Computer Architecture and Operating Systems Department, University Autonoma of Barcelona, ETSE, Edifici Q, Bellaterra, 08193, Barcelona, Spain

[b]Computer Science Department, University Catholic of Salvador, Av. Cardeal da Silva, 205, Federacão, 40220-140, Salvador, Bahia, Brazil

1. Introduction

Fault tolerance is a critical point for long-running parallel-distributed applications executing in Massive Cluster of Workstations (MCOW). From the user's point of view, a parallel application should run and finish correctly, but users rarely want to worry about including fault tolerance capabilities in their algorithms because of the software engineering costs. The cluster's administrators, in turn, request that the fault tolerance scheme consume as few resources as possible. Because increasing the number of nodes causes the MTBF to reduce, long-running applications demands a fault tolerance scheme that be independent of the cluster scalability.

The fault tolerance scheme could rely on a fault-tolerant hardware; however, such solution is expensive in practice. An alternative would be to develop fault-tolerant algorithms. However, such solution demands a big software engineering effort, and it cannot be applied to algorithms already coded. A third possibility is to build a software layer between the application and the system in order to isolate the faults from the application. This is an interesting alternative since it does not require any special hardware, and makes the fault tolerance scheme fully transparent to the user.

Rollback-recovery is the classical method used when it is necessary to offer fault tolerance for a long-running parallel-distributed application based on message passing in a cluster [7]. Rollback-recovery techniques can be implemented by a software layer, and are divided into two broad categories: the ones based only in checkpoints and the ones based in checkpoint and in message logs. The first ones make checkpoints of individual processes associated with a certain checkpoint synchronization scheme to assure that the system rolls back and recover from a consistent global state after a failure. The latter ones also aggregate message's logs for each individual process in order to allow the system to recover from a point later than the latter consistent global state [5].

The main difference between these techniques is expressed by two parameters: a) the runtime overhead over fault-free execution and b) the fault penalty. The overhead directly relates to the efficiency of the rollback-recovery scheme without faults, and it is strongly related to the cluster resources consumed by the rollback-recovery protocol [4]. For example, from the user's point of view, a fault tolerance scheme using message log interferes on the latency of message's transmission. Furthermore, both checkpoints and logs require a additional storage resources and such resources impact over the cluster architecture.

The fault penalty depends on the efficiency of the rollback-recovery scheme after a failure. It derives from two main parameters: a) the cost of recovering itself and b) the cluster architecture

*This work was supported by the MCyT-Spain under contract TIC 2004-03388 and partially supported by the Generalitat de Catalunya - Grup Recerca Consolidat 2001 SGR-00218

†The first author is thankful to the Universidade Catolica do Salvador (UCSal), Brasil, for the financial support during this work

after the failure. The cost depends on how far a set of processes must rollback in order to reach a consistent global state, i.e., how much computation is lost. Factors like process's interaction and checkpoint interval strong influence the cost of recovering. For example, short checkpoint intervals permit that a process loses little computation because it needs to recover from a relatively recent point. However, short checkpoint intervals strongly influence the system performance because of the frequent interruptions in the process's execution. The cluster architecture after a failure, in turn, determines how many nodes are available to continue the computation.

Several studies have focused on the behavior of the rollback-recovery protocols and some implementations have been built for practical systems (see section 2.5). Such studies and implementations have concentrated on the particularities of the protocols or on the performance aspects, but they gave little attention to the relationship between the fault tolerance scheme and the cluster architecture.

Such relationship is important because it allows an evaluation of the impact of the fault tolerance scheme over the cluster architecture. Furthermore, using such relationship it is possible to build models for the cluster architecture in the presence of failures. These models are very useful when the user needs to know how many failures his/her system can bear, or how failures interfere on his/her application.

Taking into consideration that any fault tolerance scheme for MCOWs must be scalable and that the relationship between the cluster architecture and the fault tolerance scheme is relevant in order to allow the user to evaluate its application in the presence of failures, we have developed a fault tolerant architecture that attends to both requisites.

Our architecture RADIC bases on two distributed arrays of dedicated processes that together implement a distributed fault tolerance controller. One array is composed by *protector* processes, which are in charge of the cluster's nodes, and the other array is composed by *observer* processes, which care of the processes of the parallel-distributed application. Both arrays of processes work transparently and independently in order to isolate faults from the application. Such architecture establishes a deterministic behavior for the cluster architecture after a node failure and it is easily adaptable to the cluster configuration.

The next section contains a description of the architecture, and a comparison with some related works. Section 3 presents the validation of the architecture using a practical implementation. In section 4, we state our conclusions and relate the future works.

2. RADIC: A scalable architecture for fault tolerance in clusters

RADIC - Redundant Array of Distributed Independent Checkpoints is a functional architecture model based on two arrays of system processes: *protectors* and *observers*. Together, these processes compose a distributed fault tolerance controller.

Protectors are processes that monitor each cluster node, and function like a distributed stable storage system. *Observers* are processes that control the checkpoints and message logs of each application process (one *observer* for each application process).

Figure 1a depicts the interaction between processes in a cluster with four nodes (N1..N4). Each node has a protector process (T1..T4). There are five *observer* process (Oa..Oe), each one attached to an application processes (A..E). Doted lines indicate the relations between *protectors* processes, and continuous lines indicate the relations between *observers* and *protectors*.

The overhead caused by RADIC over a failure-free operation is the same caused by any fault-tolerance scheme since it does not add new overheads for the recovery mechanism. The overhead on application runtime will mainly depend on three factors: a) runtime enlargement of the application processes caused by the checkpoints, b) message delaying caused by the logs and c) computation of

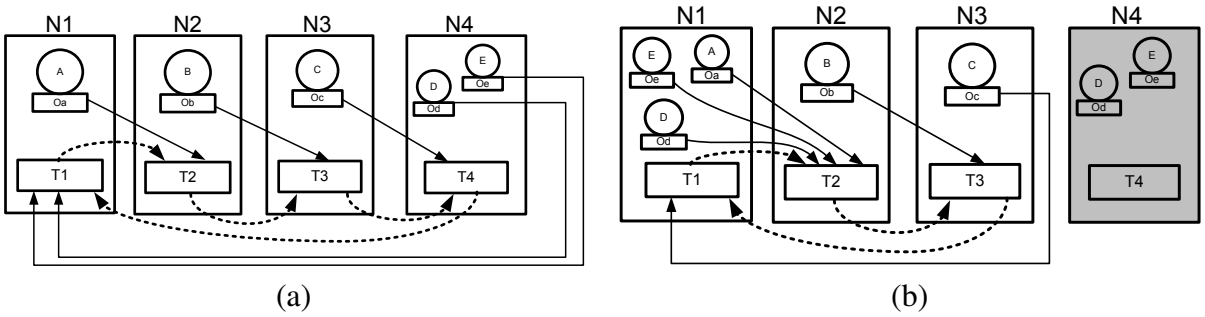


Figure 1. An application composed by 5 process (A..E) running in a cluster with 4 nodes (N1..N4) including the RADIC components. a) Operation without failures; b) A fault in N4 forces T3 to find T1, Oc connects to protector T1, and T1 recovers processes D and E.

the recover protocol algorithm. The cluster suffers a performance reduction because of: a) the increasing of disk I/O (checkpoints and logs storage) in each machine and b) the increasing of network traffic (checkpoint and log transmission from the *observers* to *protectors*).

Fault penalty depends on: a) the number of processes that will rollback; b) how much computation is lost in each application process rollback; and c) the final architecture of the cluster after a failure. The last factor increases application runtime if the final cluster architecture has fewer nodes, which causes a lower performance. Nevertheless, in a large scale cluster, loss of just a few nodes in the cluster will probably have little influence on overall performance.

In the next sections we explain the basic functionality of RADIC, a more detailed explanation about the RADIC architecture functionality and modules can be found in [6].

2.1. System Model

A distributed application consists of a set of concurrent executing processes that cooperate with each other to perform a task. The processes communicate only through message passing. There are P processes in a cluster with N nodes. In failure-free executions, all the N nodes are available. The system will support a Maximum Number of Failure Nodes (MNFN). If a node fails, it will be definitely discarded. Processes that were placed in a faulty node always recover in a different node.

The distributed application does not interact with the outside world. Therefore, received messages are the only nondeterministic event and each process is modeled as a sequence of state intervals, each one started by a received message. Execution inside each interval is deterministic.

Communication channels and process are both synchronous, i.e., whenever an element is working correctly, it always will perform its intended function in a finite and known (or predictable) time bound. Therefore, every communication channel will have a latency bound and every process will execute each of their state intervals in a time bound.

2.2. *Protectors* processes

Each *protector* communicates with another *protector* in a different node, in such a way that every *protectors* is monitored by some other *protectors*. Together, all *protectors* perform a distributed failure detector. When a protector is performing a monitoring function, it sets a watchdog for each *protector* it is monitoring. The monitored *protector* regularly sends control messages in order to reset the watchdog of its monitor. If a failure occurs in a monitored node, the watchdog of the monitor *protector* detects it, and the monitor *protector* starts the necessary actions in order to recover the application processes that were placed in the faulty node. Similarly, if a failure occurs in a monitor node, each monitored *protector* detects it because it cannot send the reset message to its monitor's watchdog. In such situation, the monitored *protector* connects to a new monitor.

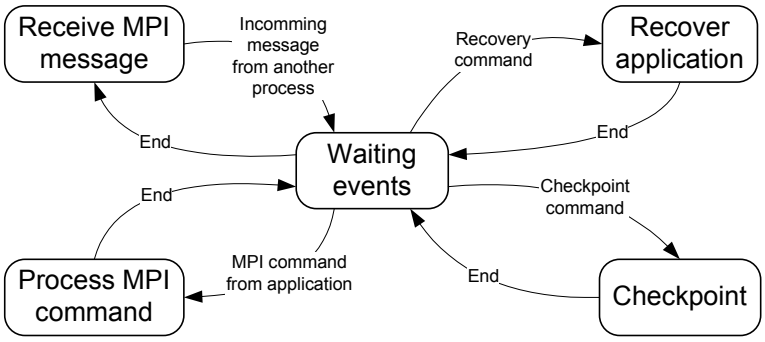


Figure 2. *Observer* state diagram

Each *protector* also communicates to a set of *observers* that are in the nodes that it monitors. For each *observer* of its set, the *protector* operates as a remote stable storage; i.e., the *protector* stores the checkpoints (and message logs) of the application processes related to each *observer* of its set. In Figure 1a, *protector* T4 monitors T3, and is monitored by T1 (a similar scheme is followed by the other *protectors*). Therefore, T4 stores the checkpoints and message logs of the application process C (via Oc) and T1 stores the checkpoints and message logs of application processes D and E (via Od and Oe).

Different *protector*'s interconnections schemes are possible. For example, the nodes can be grouped in cells regarded to protection scheme, where each cell is built by a chain of *protectors* as depicted in Figure 1a. Another possibility is to use nodes only to protect (nodes with no application process) and nodes only to compute (nodes with no protector process).

2.3. *Observers* processes

Observers are RADIC processes attached to each application processes. Each *observer* process is "owned" by an application process and has to perform several different tasks. The first task is to manage the message passing between its application process and the other processes. The second task is to maintain a mapping table indicating the location of all application processes and their respective *protectors*, i.e., in which node each application process and its respective *protector* is located. This table is updated whenever an *observer* detects a communication failure with another application process in the MPI world. Each *observer* uses a simple heuristic in order to update its table: if the communication fails, look for the process reincarnation in the node of monitor protector of the faulty process. Such heuristic avoids that the new location of a faulty node needs to be transmitted for every non-faulty node in the cluster. The third task is taking checkpoints and message-logs of its application process, and send them to the monitor protector. For non-coordinated schemes, each *observer* can have an individual checkpoint policy for its application process. Such independence allows the implementation of efficient checkpoint strategies in order to reduce the overhead caused by checkpoints.

Finally, each *observer* is responsible for performing the rollback-recovery activities when the system uses fault tolerance schemes that demands coordination during recovery. Is such cases, the *protectors* determine the which specific checkpoint is necessary for each process in order to roll back the system to a consistent state. Such checkpoints are "send back" to the respective *observer*, and the *observer* manages the roll back of their application processes.

Figure 2 shows the *observer* state diagram. The state transitions are fired by events that comes from three elements: its monitoring protector, other MPI processes and the application process related to the *observer*. There are two different events related to the monitoring protector. The first

event occurs when the *observer* takes checkpoints and message logs from its application process and sends them to the monitoring protector of the node where the *observer* is placed.

The second event occurs when the *observer* receives a recover command from the protector indicating that it should restart the application process from a previous state (for rollback recovery protocols that requires coordination).

The *observer* also has events related to other application processes in the MPI world, whenever they send messages to its application process. Finally, there are events related to application process of the *observer* whenever this process performs an MPI command.

It should be noted that for protocols that need to keep several checkpoints for each application process, the *observer* also must to maintain a copy of each checkpoint sent to the *observer*. Such requirement comes because the monitoring protector itself is unreliable. Therefore, if the monitoring protector fails the checkpoint history of their monitored processes is lost and hereafter such processes cannot be recovered.

2.4. Failure detection and recovery

Figure 1b represents a failure in node N4. In this case, T3 (monitored) and T1 (monitor) detect the failure in T4. T3 connects to monitor and finds T1. Meanwhile, because T1 was monitoring N4, T1 recovers the application processes of the faulty node N4. Simultaneously, the *observers* in the node N3 (in this case only Oc) detect that its checkpoint storage (protector T4) has failed, and search a new protector in order to establish a new checkpoint storage for their application processes.

Before a protector recovers an application process, it first determines the correct checkpoint that should be used. For protocols that require a coordinated recovery, the protector array starts a synchronization procedure in order to roll back the system to a consistent state. Since each protector has checkpoints of a subset of the application processes, they command the *observers* to resume their application processes from an earlier state in order to reach a system-global consistent state.

In order to achieve this, each protector transmits the specific checkpoint back to the respective *observer*, and commands the *observer* that restarts the application process from this checkpoint. For message log protocols, besides rolls back the system to a consistent state, the protector also replays the messages that are in the message log.

The synchronization between *protectors* is necessary only for protocols that require global coordination. The message logging pessimistic protocol does not require such coordination because the recovery data and the recovery protocol itself can rely only in local information [7].

2.5. Comparison with other solutions

There are many solutions using rollback-recovery for implementing fault tolerance in parallel-distributed systems dedicated to execute scientific long-running applications. Projects such as Starfish, CoCheck, Egida, FT-MPI, MPI-FT, LAM-MPI, MPICH-V, LA-MPI and Open MPI represent some recent efforts in attempting to incorporate network and process fault tolerance into message passing systems using checkpoint and rollback-recovery techniques.

Although such projects do present valid solutions to the fault tolerance problem in clusters, they focus on performance issues or on protocol details. They dedicate little or none attention to questions about how the cluster architecture interacts with the fault tolerance scheme or how failures influence the cluster architecture.

The RADIC architecture simultaneously attend to the following requisites: scalability, transparency, modeling of the relationship between fault-tolerance scheme and the cluster architecture, and do not request any dedicated nodes in order to operate. Therefore, in this section we have compared RADIC and other solutions taking these requisites in consideration.

CoCheck [13] relies on the Condor checkpoint library, and it is implemented on the top of tuMPI. It uses coordinated checkpoint and the recover is based on a centralized coordinator. Starfish [1] provides failure detection and recovery based on coordinated or uncoordinated checkpoint. Starfish lets the responsibility of recovering to the application. Egida [11] is a toolkit integrated with MPICH. It changes the *p4* parallel programing library send/receive functionalities, and was dedicated to compare the behavior of different rollback-recovery protocols. FT-MPI [8] is not transparent to the application. It only handles failures at the MPI communicator and the application must manage the recovery. MPI-FT [10] uses message logging together with a centralized pessimistic strategy based on a central *observer* or a distributed optimistic strategy based on each application process. In case of a failure, MPI-FT restarts a faulty process since the beginning. MPICH-V [3] does define the architecture for the cluster configuration. However, its fault-tolerant scheme relies in dedicated nodes to achieve its goal. LAM/MPI [12] uses the Berkeley Labs Checkpoint Library in order to implement a coordinated checkpoint protocol. However, it does not offer an automatic mechanism to failure detection and recovery. LA-MPI [2] focuses on network fault tolerance and does not offer fault tolerance for the application processes. Open MPI [9] is a recent MPI-2 compliant project that include fault tolerance capabilities in their implementations. Open MPI is a combination of the technologies from FT-MPI, LA-MPI, LAM-MPI and PACX-MPI. At the time this text is written, fault tolerance is still not available as a stable feature.

3. Architecture Validation

We validate the RADIC functionality using a prototype implementation called RADICMPI. This implementation includes a library (*radicmpi*) and a runtime environment (*mpicc* and *mpirun*) that facilitates the compilation and the program executions.

The current implementation of the library *radicmpi* contains the following subset of MPI functions: MPI_Init, MPI_Finalize, MPI_Send, MPI_Recv, MPI_Comm_rank, MPI_Comm_size, MPI_Get_processor_name, MPI_Wtime, MPI_Type_size. RADICMPI uses the pessimistic message log rollback-recovery protocol because it is the only that does not compromise the system scalability since it confines the effects of a failure only to the faulty process. Furthermore, this protocol simplifies garbage collection because the system can simply discard checkpoints and message logs previous to the most recent checkpoint.

Using RADICMPI, we tested the functionality of RADIC with two programs: ping-pong and matrix multiplication. Our main interest was to confirm the architecture functionality in the presence of failures. We used the matrix multiplication program in order to certify the system correctness under failure conditions. The ping-pong program was used to verify the functionality of each RADIC module and to control the injection of failures.

We ran the tests in a heterogeneous cluster with six nodes interconnected by a 100BaseT hub: 3 Athlon XP 2600+/1.9GHz/256MB (w2,w3,w4); 1 Pentium 4/2.6GHz/256MB (w5); and 2 Pentium-III/800MHz/128MB (master,w1). All nodes used Linux Fedora Core 3 with kernel 2.6.9-1.667. For checkpointing we used the library developed by Victor Zandy [14]. All softwares were compiled using GNU g++ compiler v3.4.2. The *protectors* array was organized in a chain like Figure 1a.

The overheads for a master-worker matrix multiplication algorithm are summarized in Figure 3. In order to see the impact of logs over the overall runtime, we used an algorithm that sends one matrix for all workers and then slice the other matrix among the workers. First, we run the program without any kind of protection (checkpoints or logs). Then, we executed the algorithm with all protections activated and forcing a checkpoint approximately at the middle of the execution. One can note the strong impact of logs and checkpoints over the overall performance. This impact is caused by the increasing in the message latency caused by message logging mechanism. Furthermore, the

	master	w1	w2	w3	w4	w5
250x250	141	56	133	123	104	65
500x500	58	27	67	62	50	33
1000x1000	32	41	12	11	8.3	5.4

Figure 3. RADIC runtime overheads for a matrix multiplication algorithm. The results were calculated based on a execution without protection (overhead=runtime protected/runtime unprotected).

large memory space used by the program leads to a large checkpoint overhead because of the large checkpoint storage time caused (hard disks in the protectors). The RADIC efficiency improves when the matrix increase in size because the reduction of the log impact in the whole computation time. Since we were interested only in the functionality of the RADIC architecture, we did not make any performance measure for these cases because the total application runtime after a failure depends on the moment where the failure occurs inside a checkpoint interval, because this moment determines how much a process rolls back.

4. Conclusions

We presented and described RADIC, an architecture model for implementing fault tolerance in clusters based on the concept of *observer* and *protector* processes. RADIC covers all the requirements of a fault tolerance architecture for parallel-distributed systems, and also have important features. It is scalable since it implements a fully distributed scheme for supporting faults. It is user transparent since it does not impose any change to the application algorithm. It is independent of the recover protocol, because the networks of *protectors* can store checkpoints and message logs from the *observers* in order to attend to the different strategies. It should be noted that RADIC also allow the implementation of non-scalable rollback-recovery protocols, like the ones that require global coordination in order to perform recovery. Furthermore, non-transparent fault-tolerant can also take advantage of the RADIC architecture.

We have proved the basic functionalities of the RADIC architecture making tests with RADICMPI. Now, we are developing a performance model for evaluating the total application runtime as a function of parameters like checkpoint interval; checkpoint cost; message patterns; application algorithm; RADIC organization; failure pattern; and computation/communication ratio. Our interest is to investigate how the cluster architecture is influenced by failures. Our intention is to build a model in order to allow that the user either evaluates the impact of the fault-tolerance scheme over his/her application or determine which cluster architecture should attend to his/her application runtime requisites.

We continue the development of RADICMPI in order to make it more efficient for measuring the parameters necessary for building and validating the performance model. We are also improving RADICMPI in order to use it with the NAS benchmark. Furthermore, we are evaluating the viability of using RADIC with implementations like OpenMPI or MPICH2. Such possibility would greatly facilitate new experiments.

We are also interested in evaluating how the different RADIC configurations operates in massive clusters. Since scalability is one of our main goals, we are interested in studying how the efficiency of the protection is affected by the number of machines in the cluster. Furthermore, since massive clusters can be constructed with different types of network topology, we are concerned about methods for distributing the *observers* taking into consideration their distances to the *protectors*.

References

[1] A.M. Agbaria and R. Friedman. Starfish: fault-tolerant dynamic MPI programs on clusters of workstations. In *Proc. of 8th Inter. Symp. on High Perf. Dist. Computing*, pages 167–176, August 1999.

[2] R.T. Aulwes, D.J. Daniel, N.N. Desai, R.L. Graham, L.D. Risinger, M.A. Taylor, T.S. Woodall, and M.W. Sukalski. Architecture of LA-MPI.

[3] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov. MPICHV: Toward a scalable fault tolerant MPI for volatile nodes. In *Proc. of SuperComputing 2002 (SC2002)*, November 2002.

[4] A. Bouteiller, P. Lemarinier, G. Krawezik, and F. Cappello. Coordinated checkpoint versus message log for fault tolerant MPI. In *In Proc. of 2003 IEEE International Conference on Cluster Computing*, pages 242–250. IEEE, December 2003.

[5] K.M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

[6] A.A. Duarte. RADIC: Redundant array of distributed independent checkpoints. Master's thesis, Universidad Autónoma de Barcelona, Departamento de Arquitectura de Computadores y Sistemas Operativos, ETSE, Bellaterra, 08193, Barcelona, Spain, July 2005.

[7] E.N. Elnozahy, L. Alvisi, Y.M. Wang, and D.B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[8] G. Fagg and J. Dongarra. FT-MPI: Fault tolerant mpi, supporting dynamic applications in a dynamic world. In *Euro PVM/MPI User's Group Meeting 2000*, pages 346–353, Berlin, Germany, 2000. Springer-Verilag.

[9] E. Gabriel, G.E. Fagg, G. Bosilca, and et al. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proc., 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[10] S. Louca, N. Neophytou, A. Lachanas, and P. Evripidou. MPI-FT: Portable fault tolerance scheme for MPI. *Parallel Processing Letters*, 10(4):371–382, 2000.

[11] S. Rao, L. Alvisi, and H. Vin. Egida: An extensible toolkit for low-overhead fault-tolerance. In *Proc. of IEEE Fault-Tolerant Computing Symposium (FTCS-29)*, Madison, WI, June 1999.

[12] S. Sankaran, J.M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. In *Proc. of LACSI Symposium*, Sante Fe, New Mexico, USA, October 2003.

[13] G. Stellner. Cocheck: Checkpointing and process migration for MPI. In *Proc. of Inter. Parallel Processing Symp.*, pages 526–531, Los Alamitos, CA, USA, April 1996. IEEE Computer Society Press.

[14] V. Zandy. Ckpt - a process checkpoint library. http://www.cs.wisc.edu/ zandy/ckpt/, April 2005.

An Automated Approach to Improve Communication-Computation Overlap in Clusters

Lewis Fishgold[a], Anthony Danalis[a], Lori Pollock[a], Martin Swany[a]

[a]Department of Computer and Information Sciences, University of Delaware, Newark, DE 19716
{fishgold,danalis,swany,pollock}@cis.udel.edu

Applications that execute on parallel clusters face scalability concerns due to the high communication overhead that is usually associated with such environments. Modern network technologies that support Remote Direct Memory Access (RDMA) can offer true zero copy communication and reduce communication overhead by overlapping it with computation. For this approach to be effective though, the parallel application using the cluster must be structured in a way that enables communication computation overlapping. Unfortunately, the trade-off between maintainability and performance often leads to a structure that prevents exploiting the potential for communication computation overlapping. This paper describes a source-to-source optimizing transformation that can be performed by an automatic (or semi-automatic) system in order to restructure MPI codes towards maximizing communication-computation overlapping.

1. Introduction

Clusters of workstations are in common use among engineers and domain scientists due to their high processing power to cost ratio. The major drawback of cluster-based parallel computing as compared to shared memory multiprocessors is the network delay induced by the node interconnecting technology of clusters. Several interconnection technologies such as Myrinet, Quadrics and Infiniband can improve cluster message-passing performance by providing specialized low latency, high bandwidth networks for clusters. Such technology can theoretically reduce communication latency by overlapping communication with computation through handling network traffic solely on a network co-processor, freeing the CPU to perform useful computations.

Unfortunately, many existing scientific applications follow a modular structure where the computation is separated from the communication. Although such an approach makes the code easier to maintain and alter, it prevents communication-improving network technology from being fully utilized.

To overcome the restrictions imposed by such overlap-naïve code, a program can be transformed so as to aggressively send data as soon as it is generated. In particular, the computationally expensive part of many scientific applications consists of a loop (commonly with multiple levels of nesting) that executes some basic computation kernel. The suggested transformation aims to achieve "pre-pushing" by performing the communication within the computation loop using non-blocking, asynchronous I/O operations to transfer data elements among the parallel tasks as soon as it is safe. To evaluate the potential of this transformation, Danalis et al. [3] transformed potentially benefiting applications manually and experimented with the resulting variations to study the performance gains. Their results show that near maximum communication-computation overlap can be achieved, resulting in reduction of the communication overhead and significant performance improvement in comparison to the original code, as shown in Figure 1.

Although the suggested pre-push transformation can be performed by an experienced programmer, there are several reasons to build an automated system.

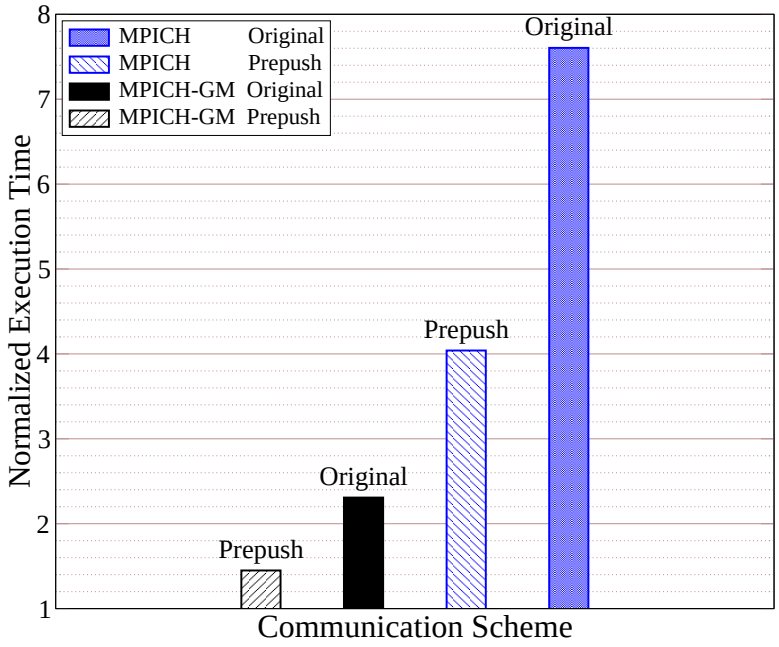


Figure 1. Performance improvement achieved by "pre-pushing"

- Asynchronous communication can be error prone and difficult to program, particularly when many processors and corresponding outstanding messages are involved creating a need for explicit synchronization.
- The performance of the transformed code depends on several cluster and application related parameters. These parameters have to be recomputed (or rediscovered through extensive profiling) every time the code changes, or the cluster CPUs, memory, or network changes.
- The suggested transformations have a negative impact on the maintainability of the code and in the case where low level communication primitives (eg., Myrinet's GM) are used, portability is also affected.
- Having an automated system perform the transformation opens the optimization to a wider audience of applications such as legacy codes, and those whose programmers are unaware of the details of the optimization.

Significant research has focused on optimizing communication latency in cluster environments but none can handle explicitly parallel codes written using MPI. Many compiler or language-based techniques translate higher-level parallel constructs into message passing primitives as appropriate. Examples of this include UPC [4], Co-Array Fortran [11], HPF [6], and Fortran-D [7]. While these approaches allow programmers to write their code in SPMD style, they focus on parallel optimization in the large, rather than focusing on optimization of messaging on a single host and do not deliver the performance that can be achieved by carefully tuned, manually parallelized applications. Systems such as Polaris [2] and PARAMAT [9] perform source-to-source transformations to achieve parallelization of serial programs that are written in *Fortran 77* or *C* without any special annotation. Nevertheless, these systems do not accept input code already parallelized with the use of MPI, but rather expect code written as a serial program. Projects such as CC-MPI [8] attempt to extend the

standard MPI in order to provide support for the *compiled communication* model [16]. In this way, communications that lend themselves to static analysis can be separated from those which do not, and optimizations can be performed as appropriate. The main difference between our project and the alternative solutions is that we aim to offer a complete system able to automatically (or semi-automatically) restructure parallel applications, explicitly parallelized with the use of MPI, in order to minimize their communication overhead by performing communication-computation overlapping.

2. Communication-Computation Overlapping Transformation

The modular structure of many scientific codes, in which some data is computed, stored in an array, and then sent over the network, leaves no opportunity for communication-computation overlap. Often, as soon as the data is ready to be sent, it needs to be used (by the receivers). We propose a transformation for such codes so that the data is pre-pushed, or sent as it is generated, before it is needed.

To achieve such an early transfer model, the computation loop is restructured into blocks, or tiles, in which each tile executes only part of the iteration space and therefore performs only part of the original computation. Consequently, each tile generates only a subregion of the original array and depending on the data dependencies of the loop, it could be the case that at the end of the tile execution, the generated array subregion is not altered by future iterations (i.e., consecutive tiles). In addition, asynchronous send and receive operations are inserted at the end of each tile so that the transfer of the array subregion generated by the corresponding tile is initiated. This transfer is completed by the network co-processor, while the CPU continues computing the next tile of the array. In order for such a transformation to preserve the correctness of the original code, the subject application needs to first be analyzed. In general, to restructure code to pre-push the results of its computation, we must first determine the following information:

- the communication operations in the original code and the corresponding computation loop(s) that write(s) to the array being sent
- the pairs of matching send and receive operation(s), since both the send and the receive must be transformed in concert
- the earliest execution point where it is safe to receive pre-pushed data (This is important in cases where the receiver uses the receive array prior to the part of the code we are trying to transform. In such a case, it is only safe to transfer the data after the latest point of such use.)

The last two are difficult to determine and in some cases they can be statically undecidable. Therefore, our transformation effort focuses on cases that reveal more information about the communication and do not exhibit such undecidability. Such information is statically known in the case of collective communication operations such as `MPI_ALLTOALL`. In such operations, both sending and receiving is implemented internally and the function call appears as an atomic, or indivisible, operation at the level of the application. In addition, the semantics of `MPI_ALLTOALL` require that all participating nodes have to call it. Therefore, we do not need to match statically the senders to the receivers; we know that all nodes exchange data, and they do so in a predetermined pattern. Regarding the computation, our current analysis focuses on computation loops where every node executes the same code. In other words, there can be no branches (i.e., `if` statements) in the code that stores data into the array that is being exchanged. Many scientific codes contain frequently executed sections consisting of a multiply-nested loop in which the inner loops execute some computation kernel

```
integer A_s(1:NX)
integer A_r(1:NX)
...
do iy=1, NX  !outer loop
   do ix=1, NX  !inner computation loop
      ...
      A_s(ix) = ... !RHS is not array ref.
   enddo
   !sends A_s and receives into A_r
   call collective-comm(A_s,A_r)
enddo
```

(a) Before

```
integer A_s(1:NX)
integer A_r(1:NX)
...
do iy=1, NX  !outer loop
   do ix=1, NX  !inner computation loop nest
      A_s(ix) = ...
      !wait for comm of prev.  tile to complete
      if(ix mod K == 0) then
         to=...
         size=K
         call async-send(A_s(...),size,to,...)
         call async-recv(A_r(...),size,from,...)
      endif
   enddo
enddo
```

(b) After

Figure 2. Abstract target code segment before and after transformation

and store the results in an array which is then exchanged using `MPI_ALLTOALL` at the end of each iteration of the outer loop (see Figure 2(a)). This communication-computation pattern is the domain on which our current transformation is focused. Sorting, LU Factorization, Finite differences, and multi-dimensional FFT constitute examples of algorithms that could fit this abstract form, and can be transformed to exploit pre-pushing.

To demonstrate the result of the transformation, Figure 2 shows an abstract target code before and after being transformed. The tiling of the computation loop nest is controlled by the parameter K which sets the number of iterations of the tile loop per tile. Determining the optimal tile size is not a trivial task, and is best performed by an automated system, since the value may change as applications migrate across platforms. However, finding the optimal value for K is beyond the scope of this paper. A discussion about the issues related to the performance critical parameters can be found in [3].

3. Automated Transformation Technique

3.1. Opportunities for Transformation

The first step toward modifying the code is identifying opportunities for transformation. To do so, the following information needs to be collected:

- $\mathcal{C}$, a call to `MPI_ALLTOALL`.
- $\mathcal{A}_s$, the array sent by $\mathcal{C}$, which is the first argument to $\mathcal{C}$.
- $\mathcal{A}_r$, the array received by $\mathcal{C}$, which is the fourth argument to $\mathcal{C}$.
- ℓ, the loop nest executed by all nodes, which finalizes all elements in $\mathcal{A}_s$ before $\mathcal{C}$ is called. ℓ is the last loop nest not in a conditional statement, lexically preceding $\mathcal{C}$, that mutates $\mathcal{A}_s$. $\mathcal{A}_s$ can be mutated directly by assignment, or indirectly by passing $\mathcal{A}_s$ by reference to a called procedure. In the former case, if the source code for the procedure is unavailable, it cannot be guaranteed that $\mathcal{A}_s$ is written. To resolve this uncertainty, the user must be queried (making the system semi-automatic), but if ℓ is the only loop preceding $\mathcal{C}$, then it is a conservative assumption to consider ℓ to be a mutator.

3.2. Compute-Copy Pattern

For each transformation opportunity, we determine the pattern by which values are computed and copied into $\mathcal{A}_s$. We currently consider two cases:

direct $\mathcal{A}_s$ is the LHS of an assignment statement where the RHS is not an array reference, as seen in Figure 2. Section 3.3 describes how to analyze codes fitting this pattern.

indirect In this case, the contents of $\mathcal{A}_s$ are computed indirectly in the sense that they are first computed in a procedure, $\mathcal{P}$, which stores them in a temporary array, and are then copied to $\mathcal{A}_s$ afterwards. As in Figure 3(a), $\mathcal{A}_s$ appears on the LHS of an assignment where the RHS contains a reference to a different array, $\mathcal{A}_t$. Each call computes a portion of the final results and writes them to $\mathcal{A}_t$ which is passed by reference. After the call to $\mathcal{P}$, the contents of $\mathcal{A}_t$ are copied to $\mathcal{A}_s$ in a copy loop, ℓ_{cp}. The purpose of this pattern is to aggregate the partial results computed by each call to $\mathcal{P}$ so that they can be sent together at the end of ℓ. The goal of Section 3.4 is to remove ℓ_{cp}, and directly send the contents of $\mathcal{A}_t$, as this avoids the copy and is thus more efficient.

3.3. Handling the Direct Pattern

If $\mathcal{A}_s$ is written directly, we first determine which parts of the send array, $\mathcal{A}_s$, can be safely sent at a given point in the iteration space of ℓ. If an array reference, $\mathcal{A}_2$, overwrites elements previously written by another array reference, $\mathcal{A}_1$, then those elements are unsafe to send between $\mathcal{A}_1$ and $\mathcal{A}_2$. Using array dependence analysis to find output dependences [15], we determine whether the element referenced by a given array reference is safe to send after that reference is reached during execution. A safe array reference, denoted $\mathcal{A}_s^f$, is one with no output dependences on it and represents the latest element to be finalized.

We could transform the program so that elements referenced by $\mathcal{A}_s^f$ are sent one at a time, as each is computed. Although correct, it is desirable for reasons of efficiency to aggregate these single element sends into fewer, larger send operations. *Array access analysis* [12] can enable this aggregation, by determining the region of $\mathcal{A}_s$ written during a single tile. To simplify our prototype implementation, we use the simplest, most course-grained access representation, known as a partial triplet, which contains the symbolic upper and lower bound of an index expression, i_k, denoted as $u(i_k)$ and $l(i_k)$ respectively. This analysis determines the size, denoted $size$, of the blocks of contiguously accessed array elements, or *blocks*, written during the runtime of K iterations of ℓ, and the offsets of the blocks, denoted $offsets$.

Using the results from this analysis, a communication loop nest can be generated to iterate through all $o \in offsets$ in order to initiate the appropriate asynchronous communication calls to transmit the generated *blocks*. Note that if the array access pattern is regular, all the data might be in just one continuous block. This is the optimal case, as the transfer of the data can be performed with a single transfer, achieving minimal overhead and high bandwidth.

3.4. Handling the Indirect Pattern

In the case that $\mathcal{A}_s$ is written indirectly, as in Figure 3, a copy loop, ℓ_{cp}, aggregates temporary results into $\mathcal{A}_s$, which will be sent once all computation has concluded. Since we want to send results as they are generated in order to overlap communication with computation, this aggregation is unnecessary. Therefore, we can directly send the contents of $\mathcal{A}_t$, which can reduce runtime by eliminating the time taken to copy $\mathcal{A}_t$ to $\mathcal{A}_s$. The flow of data from $\mathcal{A}_s$ to $\mathcal{A}_r$ can be represented as $\mathcal{A}_t \xrightarrow{copy} \mathcal{A}_s \xrightarrow{send} \mathcal{A}_r$. By transitivity, we can eliminate the copy and still complete the same operation by the equivalent $\mathcal{A}_t \xrightarrow{send} \mathcal{A}_r$.

```
integer A_s(1:10,1:10,1:10)
integer A_t(1:100)
do iy = 1, 10  !loop nest ℓ
   call P(..., A_t)
   do ix = 1, 100  !ℓ_cp
      tx = ix % 10
      ty = ix/10
      A_s(tx,ty,iy) = A_t(ix)
   enddo
enddo
```

(a) Before

```
integer A_s(1:10,1:10,1:10)
integer A_t(1:100)
do iy = 1, 10  !loop nest ℓ
   call P(..., A_t)
   call async-send(A_t(...),...)
   call async-recv(A_r(...),...)
   enddo
enddo
```

(b) After

Figure 3. Abstract indirect pattern code segment before and after removing the redundant copy

To determine the region of $\mathcal{A}_t$ that has been finalized during a tile, we cannot directly analyze the loop that wrote to $\mathcal{A}_t$ since it is inside a procedure with source code that is unavailable. Therefore, we have to infer the access pattern of $\mathcal{A}_t$ indirectly by analyzing ℓ_{cp}. It is reasonable to assume that the region of $\mathcal{A}_t$ accessed when being copied to $\mathcal{A}_s$ is the one that is finalized by one call to the procedure. In addition, if ℓ_{cp} is executed more than once per tile, which is usually the case, we need to aggregate the temporary results, but not to the degree that they were originally aggregated. To achieve this, we expand the capacity of $\mathcal{A}_t$ by adding an extra dimension, and modify the reference to $\mathcal{A}_t$ that is passed to the procedure accordingly. Finally, the resulting communication code must preserve the original mapping from $\mathcal{A}_t$ to $\mathcal{A}_s$ that was induced by ℓ_{cp}. In other words, the blocks of $\mathcal{A}_t$ must be sent to $\mathcal{A}_r$, in the same order that blocks of $\mathcal{A}_t$ were copied to blocks of $\mathcal{A}_s$. Further details for removing the redundant copy are beyond the scope of this paper, but can be found in [5].

3.5. Communication

In this paper, we focus on transforming communication using `MPI_ALLTOALL` [10], which divides arrays into NP partitions along the last dimension, each corresponding to a different node. To preserve the semantics and efficiency of `MPI_ALLTOALL`, we must ensure that data is written for each of the nodes to receive during every tile. We can guarantee that the entirety of the last dimension is traversed if the loop inducing the traversal of the last dimension, the *node loop*, is not the outer loop, in which iterations are being split into tiles. If the node loop is the outer loop, we could use loop interchange [1] to exchange the outermost loop with one of the inner loops. If data dependences do not allow us to perform the interchange, the semantics of `MPI_ALLTOALL` can still be preserved by having all the nodes send to a subset of the nodes during each tile, but this is not as efficient as network congestion may ensue if all of the nodes are competing to communicate with one or a few nodes. The method for generating communication code in this case is given in [5]. In Figure 4, we show the replacement communication code that preserves the semantics and efficiency of `MPI_ALLTOALL` when the node loop is outermost.

3.6. Transforming the Program

After the previous stages of analysis are performed, we transform the program according to the following steps:

1. Insert the communication code shown in Figure 4 at the end of the body of ℓ.

```
integer A_s(...,SZ)
...
do j = 1,NP-1
   to = mod(mynum+j,NP)
   call mpi_isend(A_s(...,(to-1)*(NP/SZ)),...)
   from = mod(NP+mynum-j,NP)
   call mpi_irecv(A_r(...,(from-1)*(NP/SZ)),...)
enddo
```

Figure 4. Communication Code

2. Insert a blocking call to wait for all outstanding receives from the previous tile to complete, before the code inserted in step 1.

3. Insert code after ℓ, to exchange any leftover elements not sent by the last tile, which result from K unevenly dividing the number of iterations of ℓ (i.e., ℓ `mod` K).

4. Insert code, after ℓ and before $\mathcal{C}$ to wait for the arrival of the last blocks of data after the end of ℓ.

5. Remove $\mathcal{C}$, the original communication.

4. Implementation and Evaluation

The automated approach presented in the last section was implemented as a Fortran 90 source-to-source code transformer, called the Compuniformer, using the Nestor program transformation framework [14]. Using a source-to-source transformer, we decouple our transformation from the specifics of any particular compiler designed for a particular architecture, allowing our optimization to be complemented with traditional compiler optimizations. Nestor is a lightweight framework for implementing transformations to Fortran 90 code, providing a parser, a transformable IR, and unparser. Nestor also includes a data dependence analysis tool which uses Petit and the Omega Test [13]. At this time, some portions of the implementation are semi-automatic, in that they require some user input, due to limitations in the built-in analysis tools; future work will develop these capabilities more fully.

We have performed a preliminary evaluation of our prototype aimed at testing the correctness and performance of the transformation. The evaluation allows us to not only verify the correctness of the implementation, but also the techniques that underly it. We wrote a test program which is simple, yet tests many of the features of the transformation process. The test code exhibits the indirect computation pattern, which complicates the transformation since we remove the redundant copy loop. The test code as transformed by our system compiles and executes, producing output identical to that of the original, suggesting the correctness of our technique and implementation.

5. Conclusions and Future Work

In this paper, we presented novel techniques to automate the transformation of explicitly parallel codes to maximize communication-computation overlap. The broader impact of this work is the

performance improvement of parallel MPI codes on networked clusters, enabling more scalable application of the parallel codes to larger numbers of processors, benefiting the large community of domain scientists using such technologies. Future work includes creating heuristics to deal with some common idiosyncrasies of real-world codes, strengthening the implementation by incorporating more sophisticated program analysis, targeting other types of collective communication, and evaluating the system's performance on a variety of real-world codes, which should inform future work on extending the system's generality.

References

[1] Randy Allen and Ken Kennedy. Automatic loop interchange. *SIGPLAN Not.*, 39(4):75–90, 2004.

[2] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, W. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Polaris: The Next Generation in Parallelizing Compilers. In *Seventh Workshop on Languages and Compilers for Parallel Computing*, 1994.

[3] Anthony Danalis, Ki-Yong Kim, Lori Pollock, and Martin Swany. Transformations to Parallel Codes for Communication-Computation Overlap. *Supercomputing*, 2005.

[4] T. A. El-Ghazawi, W. W. Carlson, and J. M. Draper. UPC Specification v. 1.1. `http://upc.gwu.edu/documentation`, 2003.

[5] Lewis Fishgold. An Automated Approach to Improve Communication-Computation Overlap in Clusters. Senior Thesis. University of Delaware, 2005.

[6] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. CRPC-TR92225, Rice University, Houston, TX, 1993.

[7] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Supercomputing*, pages 86–100, 1991.

[8] Amit Karwande, Xin Yuan, and David K. Lowenthal. CC-MPI: A Compiled Communication Capable MPI Prototype for Ethernet Switched Clusters. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2003.

[9] C. Kessler and W. Paul. Automatic parallelization by pattern matching. In *Proceeding of Second Int. Conference of the Austrian Center for Parallel Computation*, pages 166–181, 1993.

[10] MPI Forum. MPI: A message-passing interface standard, v1.1. Technical report, University of Tennessee, Knoxville, June 12, 1995.

[11] R. W. Numrich and J. K. Reid. Co-Array Fortran for parallel programming. ACM Fortran Forum 17, 2, 1-31, 1998.

[12] Yunheung Paek, Jay Hoeflinger, and David Padua. Efficient and precise array access analysis. *ACM Trans. Program. Lang. Syst.*, 24(1):65–109, 2002.

[13] William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *ACM/IEEE Conference on Supercomputing*, pages 4–13. ACM Press, 1991.

[14] Georges-André Silber and Alain Darte. The Nestor library: A tool for implementing Fortran source to source transformations. In *High Performance Computing and Networking (HPCN'99)*, volume 1593 of *Lecture Notes in Computer Science*, pages 653–662. Springer Verlag, April 1999.

[15] Michael Joseph Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[16] X. Yuan, R. Melhem, and R. Gupta. Algorithms for Supporting Compiled Communication. *IEEE Transactions on Parallel and Distributed Systems*, 14(2):107–118, 2003.

I/O & Databases

QoS-aware Query Processing in Cluster-based Image Databases

Ulf Rerrer[a], Odej Kao[a]

[a]Department of Computer Science, University of Paderborn, Fuerstenallee 11, 33102 Paderborn, Germany {urerrer, okao}@upb.de

This paper presents the concept of Quality of Service (QoS) aware resource management in cluster-based image retrieval systems. First, the paper describes image retrieval using static and dynamic feature extraction. The complexity of dynamic feature extraction requires the utilisation of powerful parallel architectures in order to provide the user with reasonable response times. The global load of the parallel system is therefore influenced be a threshold between static and dynamic image retrieval and the effort spent by the applied dynamic operators in terms of quality-adaptations. We present a QoS optimiser meeting user-defined minimum and maximum quality demands of multiple queries leading to an efficient resource management.

1. Introduction

Image management systems are major components of general multimedia databases. The application areas are numerous, e.g. art galleries, museums, photo and press agencies, civil services. A standard approach for the creation and retrieval of image databases is based on the extraction and comparison of a-priori defined features. The degree of similarity of a query image with the target images is determined by calculating a multidimensional distance between the corresponding features. An acceptable system response time is achieved, because no further processing of the raw data is required during the retrieval process. However, the extraction of features based on complete images results in a disadvantageous reduction of the detail content, as solely global features like dominant colours, shapes, and textures define the similarity. Contained objects are not sufficiently considered. Therefore, methods for content-based image retrieval with dynamically extracted features are developed in order to enable a *detailed* search based on the human approach of analysing and searching images. Selected regions of interest (e.g. objects) are transformed by various algorithms for dynamic feature extraction and sought section-by-section (sliding template) in *all archived images*. Thereby, images with specific objects or persons, irrespective of the particular environment, are retrieved.

Despite the advantages of dynamic retrieval, the developed methods are only partially suitable for real-world applications. The main reason is the immense computational load during runtime leading to response times of several hours per query. In order to accelerate the processing, a parallel architecture, middleware and parallel methods for image retrieval were developed. The resulting cluster-based image database CAIRO consists of query stations with user interfaces, a master node controlling the query execution and of computing nodes for image comparisons. Performance measurements proved a nearly linear speedup and efficiency [1].

However, the growing number of users/images in the database and the complexity of queries respectively showed that the currently applied batch processing is not suitable as it leads to unacceptable long waiting times. Therefore, methods for concurrent query execution and global load optimisation were developed [2]. The next step presented in this paper considers architecture for Quality of Service (QoS) integration into the existing retrieval system.

2. QoS for Multiple Queries

The processing of multiple queries is a complex problem, because each query blocks the system for a long time. On the other hand the response time is one of the decisive criteria for the system acceptance. Therefore, a QoS concept was designed, which considers the *global* load of the system and keeps the response time within desired time intervals in overload situations [2].

One parameter to achieve this goal is related to the number of images to be processed with time-intensive dynamic operators. The reduction of the image sets to be searched is realised by adapting the threshold values for the static image retrieval. A low value for the threshold leads to a high number of images as a result of the pre-selection and a high threshold value delivers soley the most promising images.

A second parameter considers the effort spent by the applied operators for dynamic image retrieval. An optimisation process integrated in the operator adapts the searching template to the current image section under investigation, so also rotated, mirrored or affine transformed objects can be discovered. However, the majority of the regions does not contain the object, thus the optimisation wastes time. If the optimisation is restricted to a certain depth or even totally removed, then the processing will be accelerated significantly. On the other hand some objects affected by scaling, mirroring, etc. may not be found. As only a small portion of the images is considered, but the response time is significantly shortened, this compromise seems to be reasonable in overload situations.

Nevertheless the final decision on the desired retrieval precision and thus the tolerable response time can only be made by the user. Therefore, each operator is supplied with multiple quality levels, which affect the operator parameters such as step size for the sliding window or the optimisation level. These parameters are submitted to the query scheduler which plans the query execution in a way that the response time remains in the desired time scale. In following the integration of the QoS scheduling in the CAIRO [3,4] system will be briefly sketched.

3. Architecture

The user submits a query to the image retrieval system by providing a reference picture together with additional information considering an acceptable quality range (minimum and maximum quality level) and some time constraints (earliest start-time, deadline, etc.). Figure 1 shows the basic architecture and workflow of the system.

The admission control module refuses the acceptance if queries violate either resource constraints or query deadlines (if provided). If a query is refused, the system can return information to the user about how to lower the retrieval quality and time requirements of the query. Once a query is accepted, the system forwards it to the optimisation module to assign a proper resource allocation and quality level. The problem of determining an optimal schedule with optimal level settings is NP-hard. Hence, the decision making process has to be based on a good approximation.

The optimisation module holds a waiting and a running queue of queries. The quality optimiser has to fulfill several contrary goals similar to the algorithmic approaches in Q-RAM [5–10] which we modified earlier [3,4]. First the overall completion time of all queries has to be minimised and timing constraints have to be met. Further the systems aims at the highest quality level for each query and simultaneously maximise the overall utility for efficient system operation. Common levels vary from poor to excellent where a higher level leads to higher resource requirements and longer execution time. Then the resource manager starts for each query the retrieval application on each host, processing subsets of images of the database. After the last instance of the application finished, the results are merged and finally presented to the user.

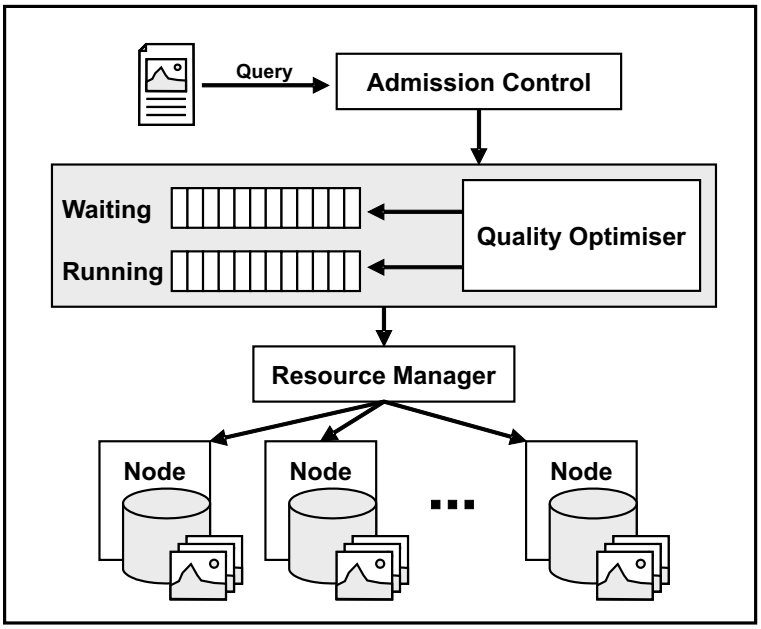


Figure 1. Architecture

4. Quality Optimiser

Every time a query is issued a new loadbalancing and level setting for the system is calculated. This is done at high level by the QoS optimiser and at a lower level by the resource manager. If a better setting could be calculated before a certain threshold and if the new setting permits the acceptance of the new query without violating any constraints, the new process schedule and quality level settings are passed to the resource manager for execution. The predicted schedule can be adjusted according to the system dependent delays ensuring the completion time of queries obeys the deadlines. The feasibility of this workaround can be found at [3].

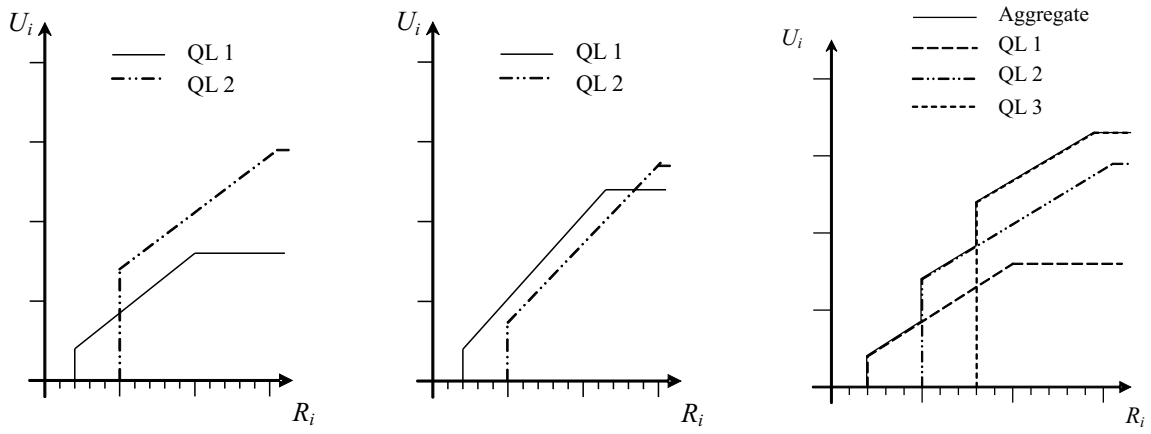
The objective of the QoS optimiser is to dynamically – i.e. with new queries (jobs) continually arriving – perform a QoS-based assignment of resources in order to maximise the system utility $U^{total} = \sum_{j=1}^{n} w_j U_j$ where n is the total number of active jobs and w_j the weight of each job utility U_j.

The utility function U_i of the job i is a measure of how well job i performs. U_i should take into account how *quickly* and at what *quality level* the job is being processed. The speediness sp can be measured by the expected time consumption with its current quality settings in comparison to the worst case estimate $\frac{t_i^{est}}{t_i^{max}}$. The expression qp measures the performance in terms of the quality level. The easiest way is to calculate the ratio of current quality level versus maximum level of quality $\frac{Q_i^{current}}{Q_i^{max}}$. However, a user might want to balance these two factors speediness (sp) and quality (qp) differently. Hence, an adequate utility function can be defined as

$$U(R) = s * sp + q * qp$$

where R is a set of resources.

When computing utilities U_i the resource allocation influences the estimated execution time of a job as well as the quality level at which it can be processed without violating its deadline. With a given number of resources, a process might also be able to reach more than one quality level. Which



(a) Aggregated with quality level as main priority

(b) Aggregated with quickness as main priority

(c) Aggregated utility for three quality levels

Figure 2. Aggregation of utility functions

option to choose depends on the weight given to the execution time versus job quality (see Figure 2(a) and 2(b)). As suggested in [5] a solution is to compute an aggregated utility. Figure 2(c) shows the aggregated utility of three jobs.

The QoS optimiser can now realise the following greedy resource allocation algorithm for a linear dimensional utility function and at least one n-ary QoS dimension.

1. Allocate minimum resources R_i^{min} to all queries p_i
2. Let R^u be the number of unallocated resources
3. For each query p_i compute slope between current utility $U_i(R_i^{min})$ and each and every utility $U_i(R)$ with $R_i^{min} < R \leq (R_i^{min} + R^u)$
4. Let p_j be the query with the largest slope s; be r the number of additional resources p_j needs to reach the Utility corresponding to s
5. Allocate r to p_j and subtract r from R^u
6. If $R^u = 0$, stop; else go to 3.

The next Section shows experimental results implementing the above algorithm. Behaviour close to real-world applications is simulated and different dynamic operators are used to get truly practicable results.

	Q-Level 1	Q-Level 2	Q-Level 3	Q-Level 4	Q-Level 5	Overhead
Algorithm 1	2	4	6	8	10	4
Algorithm 2	3	6	9	12	15	5
Algorithm 3	1	4	8	16	32	3
Algorithm 4	1	7	9	10	10	4
Algorithm 5	6	6	6	6	6	5

Table 1. Execution times of different algorithms for certain quality levels

5. Experimental Results

To evaluate the QoS optimiser and to reflect the broad spectrum of different operator behaviours we implemented five different algorithms for dynamic image retrieval. Table 1 shows the execution times at different quality levels. The utilities represent two linear, an exponential, a logarithmic and a constant time increase when the quality is increased. We decided to have five different user-selectable quality levels (*excellent*, *very good*, *good*, *acceptable* and *poor*). Less than five would be a too rough classification in the users view and more than five levels makes it difficult to imagine the significance between two levels. Table 1 also lists the overhead for each algorithm. That overhead occurs whenever its quality level is altered at run-time. Sets of images have to be transferred to other nodes and temporary results have to be created and evaluated.

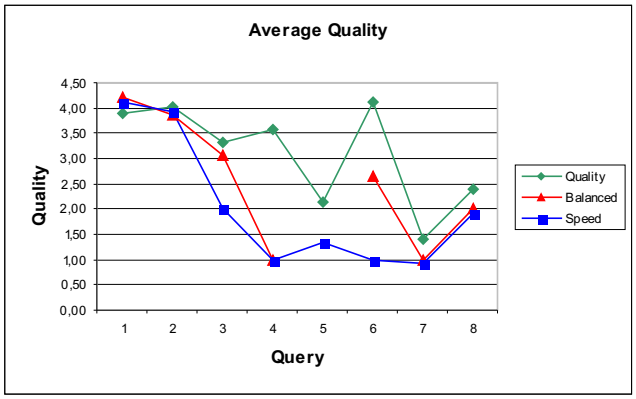


Figure 3. Average quality for eight different queries

To map a real-world situation as good as possible we set up a variety of queries using different algorithms with different deadlines. The minimal and maximal quality for each query was varied also. This set was then used in different modes of the quality optimiser. The *balanced mode* aggregates the utility function balanced between quality and speed (see Section 4). The *quality mode* doubles the quality rating in comparison to speed, and the *speed mode* vice versa.

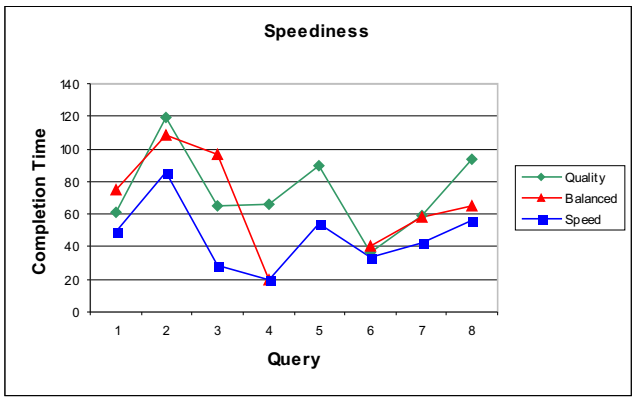


Figure 4. Speediness for eight different queries

Figure 3 shows the average quality for each of eight example queries in the three different modes. The graph shows that when the optimiser is in balanced mode the resulting average quality is between the qualities in the other modes. The average quality in quality mode is also highest in all queries.

Emphasis on the deadlines of the example queries is given in Figure 4. It shows that in speed mode the optimiser always achieves an early query completion. In quality mode the higher computational load for each query results in longer completion times due to the higher average quality.

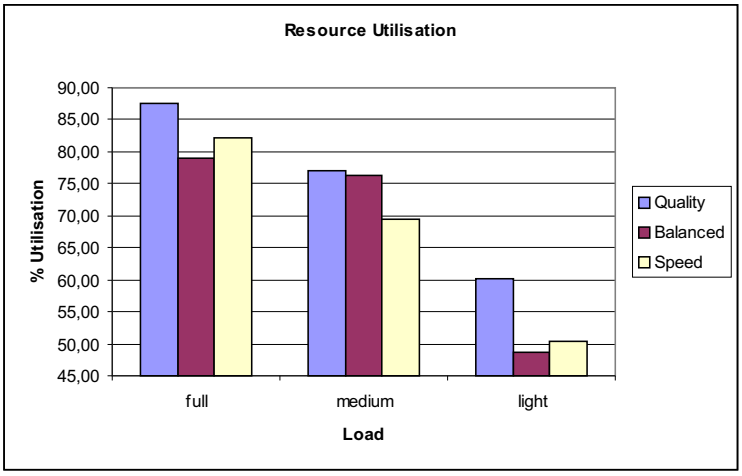


Figure 5. Resource utilisation for three different system loads

To get an impression on the systems resource utilisation and overall quality the example set of queries was changes. Three new sets of queries were defined representing a *full*, *medium* and *light* system load. The first has 18 queries each 150 time slots, the second 12 queries and for a light system load 6 queries were randomly distributed each 150 time slots.

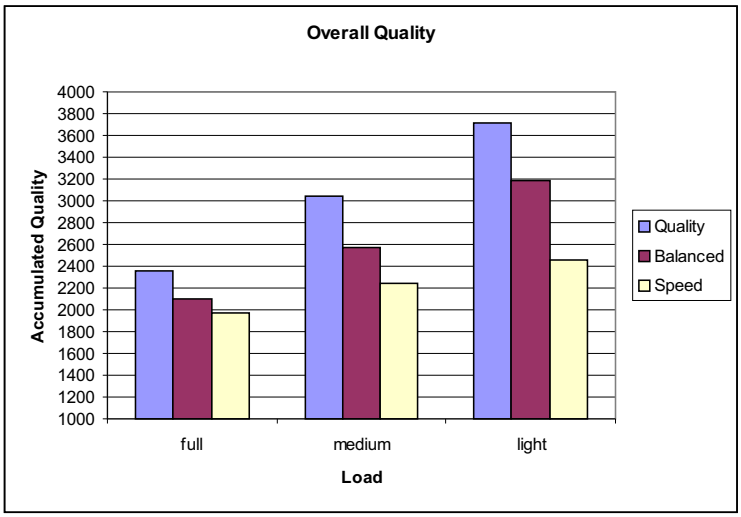


Figure 6. Accumulated quality for three different system loads

Figure 5 depicts the systems resource utilisation. The testbed had eight computing nodes which were used equally. At a full system load the resulting utilisation lies between 87 and 79 percent in all optimiser modes. In other the other load situations a lower system load occurred as intended. The highest utilisation was achieved in the quality mode at each system load.

Finally Figure 6 outlines that with more free resources it is possible to achieve a higher accumulated quality over all queries. The Figures shows again the three different load situations. The accumulated quality for all queries is highest in quality mode for all load situations as intended. In a light utilisation situation the resulting quality is almost twice as high as in a full load situation.

6. Conclusions and Future Work

Current research on the design of image databases addresses almost exclusively the increase of retrieval quality and the efficient execution of isolated queries. However, the growing number of image management systems and real fully-operational applications requires studies of efficient resource management with multiple queries. Therefore we developed a QoS-aware query processing meeting user-defined quality demands. By changing quality levels of dynamic image retrieval operators a significant load reduction is achieved. Three different optimisation modes were evaluated showing an efficient resource utilisation with high quality query results. Future work includes further investigation of different scenarios and comparison with other strategies.

References

[1] O. Kao ”On Parallel Image Retrieval with Dynamically Extracted Features” Journal of Parallel Computing, Elsevier Science, to appear 2005.

[2] S. Geisler, A. Brüning, M. Hoefer, and O. Kao ”QoS Resource Management for Cluster-Based Image Retrieval Systems” Proceedings of Parallel and Distributed Processing Techniques and Applications, to appear 2005.

[3] A. Brüning, F. Drews, M. Hoefer, O. Kao, and U. Rerrer ”Towards Quality of Service Based Resource Management for Cluster-Based Image Retrieval Systems” Proceedings of the International Conference on Algorithmic Mathematics and Computer Science (AMCS’04), 2004.

[4] F. Drews, K. Ecker, O. Kao and S. Schomann ”Strategies for Workload Balancing in Cluster-Based Image Databases” Parallel Processing Letters, World Scientific, Vol. 14, No. 1, pp. 33 - 43, March 2004.

[5] R. Rajkumar, C. Lee, J.P. Lehoczky, and D. Siewiorek ”A QoS-Based Resource Allocation Model” Proceedings of the IEEE Real-Time Systems Symposium, 1997.

[6] R. Rajkumar, C. Lee, J.P. Lehoczky, and D. Siewiorek ”Practical Solutions for QoS-based Resource Allocation Problems” Proceedings of the IEEE Real-Time Systems Symposium, 1998.

[7] R. Judd, F. Drews, D. Lawrence, D. Juedes, B. Leal, J. Deshpande, and L. Welch ”QoS-based Resource Allocation in Dynamic Real-Time-Systems” Proceedings of the 24th American Control Conference (ACC’05), January 2005.

[8] C. Lee, J.P. Lehoczky, D. Siewiorek, R. Rajkumar, and J. Hansen ”A Scalable Solution to the Multi-Resource QoS Problem” Proceedings of the 20th IEEE Real-Time Systems Symposium, pp. 315-326, 1999.

[9] J.A. Stankovic, T. He, T.F. Abdelzaher, M. Marley, G. Tao, S.H. Son, and C. Lu, ”Feedback Control Scheduling in Distributed Systems” Proceedings of the IEEE Real-Time Systems Symposium, December 2001.

[10] Y. Krishnamurthy, V. Kachroo, D. Karr, C. Rodrigues, J. Loyall, R.E. Schantz, and D. Schmidt ”Integration of QoS-Enabled Distributed Object Computing Middleware for Developing NG Distributed Allpications” Proceedings of the ACM SIGPLAN Workshop on Optimazation of Middleware and Distrubited Systems, June 2001.

A New Algorithm for Join Processing with the Internet Transfer Delays

Kenji Imasaki [a], Sivarama Dandamudi[a]

[a]School of Computer Science, Carleton University 1125 Colonel By Drive, Ottawa, Canada

This paper focuses on cluster-based parallel database systems in which only one of the nodes has the database and the other nodes, which have no initial data, are used for parallel query processing. In such a system, the load of each node changes dynamically depending on the activities of the local users. In addition, in database query processing, data skew exists. With the increasing Internet connectivity, it also becomes necessary to query databases spread around the world. In this scenario, the system can experience arrival delays and/or the transfer rate variations while receiving the join input relations. This paper investigates join processing algorithms under these conditions and proposes a new join algorithm called Symmetric Chunking Hash Join (SCHJ) that divides the hash buckets into chunks and uses them for load balancing. The SCHJ is compared with two incremental hash mapping algorithms. The experimental results conducted on a Linux cluster show that the SCHJ algorithm is the best among these algorithms.

1. Introduction

With the availability of Giga-hertz processors, Giga-byte memory, and Giga-bit bandwidth communication networks, huge parallel processing power from parallel computers can be used for various types of scientific computing. However, this power does not come for free as parallel systems are very expensive. Also, with the fast pace of technological advances, constituent components of the parallel computers quickly become obsolete. *Cluster systems* have been introduced as an alternative to such parallel systems. Database query processing can also benefit from parallel execution on such cluster systems. The query processing is managed by a Parallel Database Management System (PDBMS).

With the advent of cluster computing environments, parallel query processing on a cluster system has been proposed as an alternative to parallel database systems. In general, there are two approaches to implementing a PDBMS on a cluster system. One is the same as a traditional PDBMS: the data are de-clustered and a query is executed in parallel. Most of the recent commercial PDBMSs use this approach. The other approach is to use an existing dedicated DBMS and processing nodes (PNs) in clusters for parallel processing to take the query load off the DBMS [3,7]. We call this system a cluster-based PDBMS (*cPDBMS*) to distinguish the two approaches. This paper focuses on query processing in a cPDBMS.

When processing a query, choosing an efficient parallel query processing algorithm is important. Query processing can be improved by exploiting intra-operator (single-join), inter-operator (multiple-join), and inter-query (multiple query) parallelism [1].

Among these three types of parallelism, the single-join operation has attracted a lot of attention, since it is the most expensive operation in query processing. Hash join algorithms are clearly superior than other algorithms for the single-join operation [11]. However, hash join-based algorithms suffer from various kinds of skew [9]. Thus, the choice of load balancing/sharing algorithms becomes important since the slowest PN which has the heaviest data skew dictates the performance of the overall system.

Many researchers have proposed load sharing/balancing algorithms for the hash join algorithm [2,7,11]. Also, Hua et al. [5] compared the performance of the following load balancing algorithms on a shared-nothing parallel computer: (1) no load balancing, (2) conventional bin-packing, (3)

sampling, and (4) incremental methods. They concluded that the sampling method is the best.

These load balancing/sharing algorithms for PDBMSs on parallel computers do not work for cPDBMSs for the following reasons.

Firstly, these algorithms only deal with PNs with pre-partitioned data. However, PNs in clusters are dynamically determined and usually do not have any of the data needed for join processing. The data should be sent from DBMSs to a PN prior to the join processing. This phase is not considered in these algorithms. This situation can be seen as the extreme case of tuple placement skew [9]. In tuple placement skew, some PNs read more tuples while others wait for them to finish reading the whole relation. Secondly, these algorithms do not consider the effect of non-query background load. Even with the adaptive approach, it is difficult to know how much work a PN should transfer to another. Besides, it is not clear whether the transfer is effective or not in the case of clusters in which the load on each PN changes very frequently. In addition, no algorithm considers the effect of the combination of background load and data skew. Lastly, input data arrival delay and data transfer rate fluctuation are not considered. This is very important, especially in the case of data integration.

Therefore, a new load balancing/sharing algorithm is needed to improve the performance of join processing on clusters. This paper proposes a new load balancing/sharing algorithm for cPDBMSs. This algorithm, called Symmetric Chunking Hash Join (SCHJ), divides the hash buckets into chunks and uses them for load balancing. Also, the algorithm is based on the symmetric join algorithm, which does not distinguish between the two input relations.

2. Symmetric Join Algorithms

The symmetric hash join algorithms were proposed by Wilchut et al. [14]. Also, recently developed algorithms for data integration systems (i.e., XJoin [13]) are based on symmetric single-join algorithms. We combine symmetric hash join algorithms with the ChunkHJ algorithm proposed in [7]. In this section, we first explain the environment. Then, the load balancing/sharing algorithms are described. Next, experimental environments, including the Internet transfer delay model, are discussed.

2.1. Single-Join Processing Environment

This subsection presents the environments for symmetric single-join processing. In this environments, data is coming from remote sites to the local cluster , which consists of several processing nodes with single or dual CPUs. The local cluster is used for parallel query processing.

We developed a system to simulate query processing in the local cluster in this model. A description of each component of this system is shown in Table 1. All components are implemented by Java threads, which run concurrently on PNs. The main focus of this paper is on the JoinManager, JoinExecutors, and the Database. The JoinManager reads data from a Database and coordinates load balancing/sharing of several JoinExecutors.

With the same idea as ChunkHJ, a lot of hash bucket chunks are created using threshold values for load balancing/sharing purpose. In order to ensure the correctness of the join execution, a Join State Matrix (JSM) resides on the JoinManager is used. Each entry in the JSM represents the join status of matching pairs for each bucket.

2.2. Load Sharing/Balancing Techniques for Symmetric Hash Join Algorithms

We designed and implemented several load balancing/sharing algorithms for symmetric hash joins. The main focus is to determine hash mapping from hashId to JoinExecutorId. It is stored in *hashMappingTable* for load sharing/balancing. We developed three algorithms to decide the hash mapping. The following subsections explain these algorithms in detail. The functions used in the pseudocode description in the following subsections are summarized in Table 2. The *recv*, *send*, and *broadcast* functions are based on the MPI functions.

Class	Description
JoinMgr	makes load-balancing decision (JM)
JoinExec	executes local joins (JE)
TranExec	transfers chunks
DBRead	reads rel. and puts into buffer (DR)
DBMgr	accepts a read request and invokes DRs
Backgrd	simulates non-query process by busy looping
HashGen	uses function to generate buckets (HG)
ChunkStr	stores the chunk and deals with I/Os

Table 1
Major components descriptions.

Name	Description
recv	receives a *msg* from the master or a slave (*srcId*) with message *tag*
send	sends a *msg* with a message *tag* to the master or a slave (*destId*)
broad	broadcasts a *msg* to all slaves
apHash	applies a hash function and creates *n* hash buckets
read	reads from DBMS within the range
execLJ	executes the local join algorithm and stores results in the result buffer;

Table 2
Functions used in the pseudocode description.

The pseudocode for JoinManager is shown in Algorithm A.1. The pseudocode for HashGenerator is also shown in Algorithm A.2. *expandJSM* is used to expand the JSM entry according to the argument *info* (a pair of hashId and chunkId) sent from HashGenerator. It also fills the expanded matrix entries with "R"(Ready) entry. *FindJE* is the sender-initiated part of the algorithm that tries to find an idle JoinExecutor with the maximum number of matching pairs for this bucket. Each algorithm has a different version of *findJE* as we will explain later. *FindBucket* is used to find a suitable bucket pair with an idle JoinExecutor. For the hash mapping-based algorithm (non-SCHJ algorithms), hash mapping is used to find a suitable bucket pair. For SCHJ, the algorithm described in Section 2.3 is used. Pseudocode for JoinExecutor is shown in Algorithm A.3. *RSTransfer* is used to transfer the bucket among DB and JoinExecutor for load balancing/sharing purpose.

2.3. Symmetric Chunking Hash Join

This algorithm (SCHJ) is based on ChunkHJ [7]. Thus, hash mapping is not used. The pseudocode of *findJoinExecutor(h,cId,chunkSize)* of *SCHJ* is shown in Algorithm A.4. In this algorithm, *findSlave* [7] is used. It finds an idle slave[1] with the other matching bucket. However, the buckets which have "R" entries in JSM are considered.

Every time a JoinExecutor finishes the job, the JoinManager invokes *findBucket* (Algorithm A.5) after receiving a job request message from a JoinExecutor. *findBucket* selects a chunk pair using *findBucket(LT,chunkSize)*. Then the *findBucket(HT,chunkSize)* algorithm [7] finds a bucket in which the number of tuples is greater than the value specified in the parameter.

After a bucket chunk pair is selected, chunk transfer is done using *RSTransfer*. If at least one of the chunk pairs is not present on a JoinExecutor, then it is sent from the source node to the destination node by TransferExecutor on the source node. The source node is determined randomly.

This algorithm may perform well in the case that background load and/or data skew exist since load balancing/sharing is done in a dynamic and incremental way. However, too many transfers may cause performance degradation.

2.4. Greedy Incremental Hash Mapping and JSM-based Incremental Hash Mapping

Greedy Incremental Hash Mapping (GIHM) and JSM-based Incremental Hash Mapping (JIHM) try to deal with the Internet transfer delay that occurs in the data integration systems. The basic

[1]The terms "slave" and "JoinExecutor" are used interchangeably.

idea of these algorithms is to delay JoinExecutors to work on a bucket until there is enough work available in the bucket. The unit of work measurement is the number of tuples (GIHM) or the number of join-ready entries in JSM (JIHM).

GIHM uses a greedy algorithm and incrementally determines the hash mapping according to the arrivals of hash chunks. The pseudocode of *findJE(h)* (GIHM) is shown in Algorithm A.6. *FindBucket* uses hash mapping (omitted here). This algorithm is greedy in the sense that the JoinManager looks for an idle JoinExecutor and assigns the JoinExecutor to the hashId (hash mapping) immediately. When there are several JoinExecutors, one of them is chosen randomly as a target JoinExecutor. After the target JoinExecutor is selected, the JoinExecutor receives the hash chunk from TransferExecutor in the same way as *SCHJ* and starts the join.

In JIHM, JoinManager waits for the hash mapping assignment until the number of JSM-ready entries reaches a pre-defined number.

JIHM and GIHM may perform well when the arrival relation is delayed due to dynamic characteristics of the Internet transfer.

3. Experimental Environment

The LINUX cluster that we used in our experiments consists of 4 dual CPU nodes (Xeon 2.4 GHz and 1 GB main memory) and 7 single CPU nodes (P4 2.4 GHz and 1 GB main memory). Each node runs the Red Hat 8.0 Linux.

MPI (Message Passing Interface) [10] is now the de facto standard for programming languages for parallel processing. We use mpich 1.2.5.2 since it has thread-safe architecture and our simulation uses several threads running concurrently on PNs.

We use mpiJava 1.2.5 [4] with JDK 1.4.1 to combine the advantage of MPI and Java. MpiJava is an object-oriented Java interface to the standard MPI. MpiJava itself does not assume any special extensions to the Java language. It is portable to any platform that provides compatible Java-development and native MPI environments. We did not use pure Java parallel processing such as RMI for performance reasons.

We set memory size for a component to 24MB and node allocation sequence to interleave (single CPU, dual CPUs, single, ...) and background loop period to 1 second.

The experimental database used in the experiments consists of 1 million tuples. We created 10 relations with different random seeds for each parameter setting. In this experiments, scalar skew model [7]. In scalar skew mode, for a relation of size $|R|$, in each attribute the value 1 appears in a fixed number of tuples, while the remaining tuples contain values uniformly distributed between 2 and $|R|$.

In order to model data transfer delay over the Internet, we first measured the transfer time of different message size by UNIX ping command [12] 100 times at 3 different times of the day from our office in Ottawa to several locations spanning a range of distances from University of Toronto, University of California Los Angles (UCLA), to University of Tokyo.

Figure 1 and Table 2 show ping transfer time and approximate functions for each location.

We decided to use hypo-exponential distribution to simulate data transfer rate fluctuation over the Internet since coefficient of variation (CV) (= standard deviation/average time) is below 1. As a result, we use the following model to get the transfer times (*ActualTransferTime*) of data as a function of *size*:

- MeanTransferTime = a**size* + b (column 4 of Table 2)
- ActualTransferTime = Hypo_exponential (*MeanTransferTime,dev*) where *dev* is its standard deviation

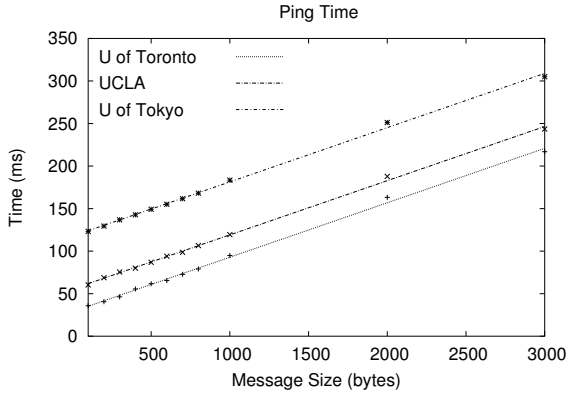


Figure 1. Ping transfer time: trend line is shown in column 3 of Table 2.

Location	Approx. Transfer Func. (trend lines in Fig. 1)
Univ. of Toronto	t=0.064×size+28.839
UCLA	t=0.0637×size+55.416
Univ. of Tokyo	t=0.0638×size+117.57

Figure 2. Approximate transfer functions.

We believe that this model is the first step to model the Internet transfer delay accurately. The statistical analysis of this model is our future work.

We insert a sleeping function after reading the relation and before applying the hash function with a duration that corresponds to the above model.

In the experiments, it is assumed that one relation resides on the local area network and the other relation resides in another location (either at Toronto, UCLA, or Tokyo). The reason for this decision is that if both of them reside on remote locations, then it is better to execute the join on one of the remote locations.

4. Experimental Results

The experimental results obtained by executing the symmetric hash join algorithms and their load balancing/sharing algorithms described in Section 2.2 are presented in this section under the data skew, background load, and the Internet transfer delay conditions. The execution is repeated until 99% confidence interval is obtained. For GIHM, we use 5, 10, and 20 as its threshold values. The complete results can be found in [6].

4.1. Performance with the Internet Transfer Delay and Data Skew

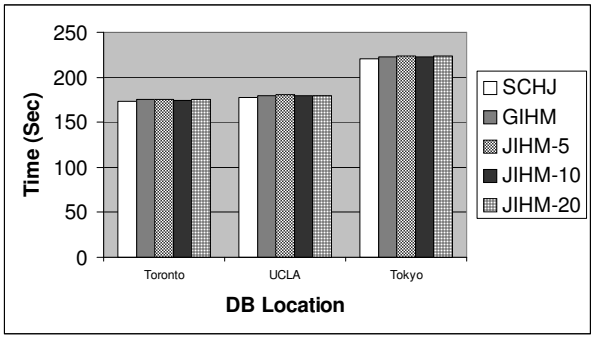


Figure 3. Performance of algorithms with scalar data skew factor of 20,000.

Figures 3 show the performance of the algorithms when there is no background load and the scalar skew factor is 20000 which changing DB location. SCHJ is marginally better than the other algorithms for all DB locations and all the degrees of skew.

In the figure, the delay caused by the Internet transfer delay from one location to another location is 4% (from Toronto to UCLA) and 25% (from UCLA to Tokyo). Table 1 shows the delay is 13% (from Toronto to UCLA) and 26% (from UCLA to Tokyo). Thus, when the delay is small the algorithms can absorb the delay. However, as the distance becomes long, the algorithms are affected by the delay.

Another interesting point is that the skew effects on the performance are not as large as the case without the Internet transfer delay [6]. This is because JoinExecutors can work on the join processing on skewed bucket while waiting for the relations to arrive as long as there is no background load on them. If there is a background load, it delays the join execution as we will see in Section 4.3.

4.2. Performance with the Internet Transfer Delay and Background Load

Figure 4 shows the performance of the algorithms as the function of background load when the DB location is Tokyo. This plot shows that the effect of the background load is small on SCHJ compared to the other algorithms because of its adaptive load balancing/sharing mechanism. Thus, the higher background load, the higher the improvement. SCHJ improvements over GIHM are 2% and 5% when the background load is 3 and 6 processes, respectively.

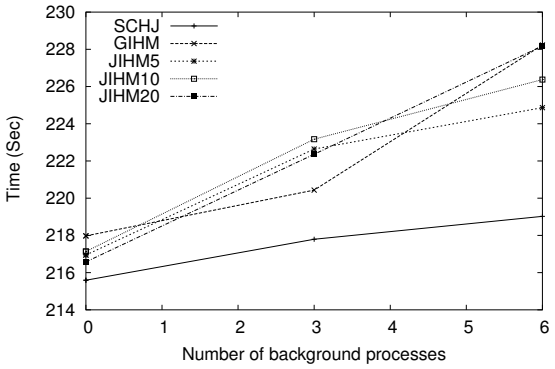


Figure 4. Performance of algorithms with the DB location as Tokyo and no skew.

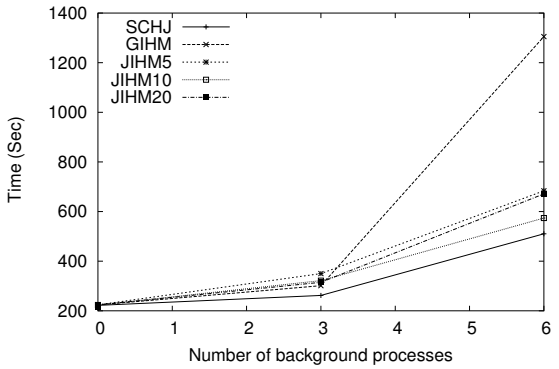


Figure 5. Performance of the algorithms with the DB location as Tokyo and the skew factor of 20,000.

4.3. Performance with the Internet Transfer Delay, Data Skew and Background Load

Figure 5 shows the performance of the algorithms as a function of the number of background load processes when the skew factor is 20000 and the DB location is Tokyo. The figure shows that SCHJ is the least affected by the background load. Thus, the higher the background load, the more SCHJ improves compared to other algorithms, which are affected by the change in background load.

Performance of SCHJ is 11% better than JIHM10, which shows the effectiveness of SCHJ. JIHM10 is better than the other JIHMs and GIHM. When the skew factor is high, the effect of the background load is severe on GIHM. GIHM is good for moderate skew case and low background load. In case of high skew and high background load, it does not perform well because of its greedy algorithm which results in poor hash mapping decision. On the other hand, JIHM waits for more data to arrive before it makes a decision. Among the various JIHM algorithms, smaller number (5 or 10) of ready JSM entries is better in these cases. If it is 20, it waits too long and keeps JEs idle long.

5. Conclusions

In this paper, we first proposed Symmetric Chunking Hash Join (SCHJ). We also proposed Greedy Incremental Hash Mapping (GIHM) and JSM-based Incremental Hash Mapping (JIHM) mainly for the Internet transfer delay case. They are compared with SCHJ for the Internet transfer delay model. In the model, one of the relations resides locally and the other resides at a remote location that has a dynamic transfer delay depending on the geographical distance. The reason for assuming a local relation is that if both of them reside on remote locations, then it is better to execute the join on one of the remote locations.

We draw the following conclusions: (1) With only the Internet transfer delay or with data skew (scalar skew), SCHJ is marginally better than the other algorithms. In this case, data skew can be absorbed by the arrival delay if there is no background load for all algorithms. (2) The greater the background load, the better the SCHJ performance because of the load balancing mechanism. (3) The improvement of SCHJ becomes greater with the increasing data skew or the background load. (4) When there is modest skew and background load, GIHM is better than the JIHM algorithms but worse than SCHJ (5) When there is extreme skew and background load, JIHM is better than the GIHM algorithm but worse than SCHJ.

References

[1] David J. DeWitt and Jim Gray. Parallel Database Systems: The Future of High-Performance Database Systems. *Communications of the ACM*, 35(6):85–98, June 1992.

[2] David J. DeWitt, Jeffrey F. Naughton, Donovan A. Schneider, and Srinivasan Seshadri. Practical Skew Handling in Parallel Joins. In *The 18th VLDB*, pages 27–40, August 1992.

[3] Matthiew Exbrayat and Lionel Brunie. A PC-NOW Based Parallel Extension for a Sequential DBMS. In *PC-NOW*, pages 91–100, May 2000.

[4] HP Java Project. mpiJava Home Page. available at http://www.javagrande.com/mpiJava.html.

[5] Kien A. Hua and Wallapak Tavanapong. Performance of Load Balancing Techniques for Join Operations in Shared-nothing Database Management Systems. *Journal of Parallel and Distributed Computing*, 56(1):17–46, January 1999.

[6] Kenji Imasaki. *Parallel Query Processing on a Cluster-based Database System*. PhD thesis, School of Computer Science, Carleton University, September 2004.

[7] Kenji Imasaki and Sivarama Dandamudi. An Adaptive Hash Join Algorithm on a Network of Workstations. In *IPDPS*, April 2002.

[8] Holger Märtens. Skew-Insensitive Join Processing in Shared-Disk Database Systems. In *International Workshop on Issues and Applications of Database Technology*, pages 17–24, July 1998.

[9] Holger Märtens. A Classification of Skew Effects in Parallel Database Systems. In *European Conference on Parallel Computing (EURO-PAR)*, pages 291–300, Manchester, United Kingdom, August 2001.

[10] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. University of Tennessee, Knoxville, Tennessee, June 1995.

[11] Donovan A. Schneider and David J. DeWitt. A Performance Evaluation of Four Parallel Join Algorithms in a Shared-nothing Multiprocessor Environment. *SIGMOD Record*, 18(2):110–121, June 1989.

[12] W. Richard Stevens. *UNIX Network Programming; Volumne I, Networking APIs: Sockets and XTI, Second Edition*. Prentice Hall PTR, 1998.

[13] Tolga Urhan and Michael J. Franklin. XJoin: Getting Fast Answers From Slow and Bursty Networks. Technical Report CS-TR-3994, University of Maryland, College Park, February 1999.

[14] Annita N. Wilschut, Peter M. G. Apers, and Jan Flokstra. Parallel query execution in PRISMA/DB. In *Proceedings of Parallel Database Systems*, pages 424–433, September 1991.

A. Algorithms

Algorithm A.1: JOINONJOINMANAGER(cS)

{Input: cS for chunk size; Output: none}
$send(HashGenerator$, "RelationRead", "R")
$send(HashGenerator$, "RelationRead", "S")
repeat
  $recv(source, tag, info)$ {info:hashId/chunkId}
  if (*tag* is "JSMUpdate") {from HG}
    then $expandJSM(info)$
      $JEx \leftarrow findJE(info, cS)$;
        {different for each algorithm}
      $send(src$, "JSMUpdateReply", $JEx)$
  else if (tag is "JobRequest"){from JE}
    then change corresponding completed JSM entries to "F"
      $chunkPair \leftarrow findBucket(cS)$
      $RSTransfer(chunkPair)$
until ((finished reading relations(R and S)) and (JSM entries are all "F"))
$broadcast$("ProcessEnd", $null$){to all JEs/HGs}

Algorithm A.2: JOINONHG(X, pS, cS)

{applies hash function on tuples of X}
{Input: relation X;
pS for partition size for relation X;
cS for chunk size; Output: none}
repeat
  $recv(JoinManager$, "RelationRead", $X)$
    {Read relation X by Database and DatabaseReader}
  $nPartitions \leftarrow |X|/pS$
  for $i \leftarrow 0$ **to** $nPartitions$
    do $X^i \leftarrow read(X, i * pS, (i + 1) * pS - 1)$
      {actual reading is done by DatabaseManager}
      $X^i_{ALL} \leftarrow applyHash(X^i, cS)$
until $recv(JoinManager$, "ProcessEnd", $null)$

Algorithm A.3: JOINONJE()

{Symmetric Hash Join on a JoinExecutor}
{Input: none; Output: none}
repeat
  $recv(TransferExecutor$, "Relation", $X_h)$
  if(X is R)
    then $execLJ(X_h, S_h)$
    else $execLJ(R_h, X_h)$
  $send(JoinManager$, "JobRequest", $null)$
until $recv(JoinManager$, "ProcessEnd", $null)$

Algorithm A.4: FINDJE$_{SCHJ}(h, cId, cSize)$

{find a JoinExecutor(JEx) using Chunk HJ}
{Input: h for hashValue; cId for chunkId; cSize;
Output: destination JE index}
$SLx \leftarrow findSlave$("LT", h) {shown in [7]};
{LT : SLx should have other paring bucket.}
if (SLx is null)
  then $SLx \leftarrow findSlave$("HT", h)
{HT : SLx does not need the other pairing bucket.}
return (SLx)

Algorithm A.5: FINDBUCKET($cSize$)

{Find suitable bucket chunk pair from JSM Entry}
{Input: chunkSize;
Output:bucketPair(hashId,chunkId1,chunkId2)}
$bucketPair \leftarrow findBucket$("LT", $cSize$)
    {find local hash bucket but use JSM [7]}
if (bucketPair is null)
  then $bucketPair \leftarrow findBucket$("HT", $cSize$)
return $(bucketPair)$

Algorithm A.6: FINDJE$_{GIHM}(h)$

{Decide destination JoinExecutor}
{Input: h hashValue;
Output: destination JE index
if (hashMappingTable.containsKey(h))
  then return $(hashMappingTable.get(h))$
  else $x \leftarrow$ random selection from idle JE list
    $hashMappingTable.insert(h, x)$
    return (x)

Comparing Two Parallel File Systems: PVFS and FSDDS

Jorge Buenabad-Chávez[a], Santiago Domínguez-Domínguez[a]

[a]Sección de Computación, CINVESTAV, Apartado Postal 14-740, México D.F., 07360, México.

1. Abstract

Parallel file systems are used to improve the performance of out-of-core parallel applications. The Parallel Virtual File System (PVFS) uses caching both to improve performance and to support logical views of data in order to simplify programming. The file system atop the data diffusion space (FSDDS) maps files onto an all-software distributed shared memory, and thus implicitly uses a relatively large cache. In this paper, we contrast the programming interface of PVFS and FSDDS and present some experimental results on their performance using two applications.

2. Introduction

Parallel file systems stripe file data across different I/O nodes so that data can be accessed in parallel on multiple, concurrent read/write operations to the same file. They are essential to improve the performance of out-of-core applications. First designs were targeted at massively parallel systems. Today the ubiquity of PC clusters and the availability of open/free designs and of other software tools have made their use and research wide-spread [2,3,5,7,9,11].

Parallel data access is a key factor for good performance, but is not the only one. Different applications manage different data structures in different ways [10,12]. In parallel applications, this implies a logical data partitioning among the processors which may, or may not, match the physical data partitioning (striping) of data across I/O nodes by a parallel file system [13]. This mismatch tends to increase the number of I/O operations, resulting in poor performance. Some I/O interfaces reduce the number of I/O operations through caching: reading/writing larger amounts of data than that requested by applications. Some interfaces also allow applications to specify a logical view of the data and access it accordingly. On a read, the interface reads all the data blocks that contain the logical data (possibly in parallel from different I/O nodes), shuffles the data according to the logical view and delivers it to the application. This also requires caching.

In this paper we compare two parallel file systems: PVFS and FSDDS. PVFS (Parallel Virtual File System) was designed for Linux clusters [2], and has gained general acceptance. It can be used with the MPI-IO interface which allows applications to access data according to logical views.

FSDDS stands for File System atop the Data Diffusion Space (DDS) [4]. DDS is another all-software distributed shared memory, and FSDDS supports file mapping onto its shared address space. DDS manages some memory as a cache in each node to dynamically map shared data; FSDDS thus benefits of a relatively large cache.

In this paper we contrast the programming interface and present some experimental results on the performance of PVFS and FSDDS. In Sections 3 and 4 we present background to PVFS and FSDDS, respectively. In Section 5 we compare their performance running 2 parallel applications on different processor counts. We conclude in Section 6.

3. The Parallel Virtual File System

The Parallel Virtual File System (PVFS) was designed for Linux clusters. It supports several APIs to access PVFS files, can be mounted as a UNIX file system (*ls*, *cp* and *rm* commands work on PVFS files), and is rather easy to install and use [2]. It is open/free software.

File data in PVFS is striped across different I/O nodes based on three metadata parameters: *pcount* specifies the number of I/O nodes across which data is striped; *base* specifies the I/O node where the striping begins; and *ssize* specifies the stripe size. PVFS handles default values for the striping metadata, which the user can change for each file. File data and metadata are stored in files in the local file system in each node, both for simplicity and for portability.

PVFS is organised as a client/server system. Server nodes have hard disk space and are those across which file data is striped; they are called I/O nodes. Client nodes are those where application processes run, issuing read/write requests to I/O nodes; they are called compute nodes. PVFS software allows each node to be either a compute node or an I/O node, or both. Application processes are linked to a PVFS library which allows them to communicate with I/O nodes through TCP.

3.1. PVFS APIs

PVFS files can be accessed with different APIs: a native PVFS API, the UNIX/POSIX API [6], and the MPI-IO interface [8]. The native API offers similar functions to the UNIX one for accessing files. It also supports functions to access noncontiguous data in a file in a single call. The MPI-IO interface offers typical functions for accessing a file, but also offers functions to define logical views, and collective I/O functions which may, or may not, be based on a logical view.

Figure 1 shows in C language code the main points involved in the definition and use of a logical data view using MPI-IO. The logical view is that, for an $N \times N$ matrix and p processors, processor 0 uses only the first N/p elements in each row, processor 1 uses only the second N/p elements in each row, and so on. (This view is useful to implement a matrix multiplication algorithm, $C = A \times B$, where each processor computes a partial value of each element in matrix C; before reading A, each processor will have read N/p rows of matrix B; see Section 5.)

```
MPI_Datatype newtype;                                  /* the new logical view */
int ndims=2,

array_of_gsizes[0]   = N;                              /* size of each dimension*/
array_of_gsizes[1]   = N;
array_of_distribs[0] = MPI_DISTRIBUTE_BLOCK;           /* divide rows by block */
array_of_distribs[1] = MPI_DISTRIBUTE_NONE;            /* do not divide columns */
array_of_dargs[0]    = MPI_DISTRIBUTE_DFLT_DARG;       /* block = rowsize/processors */
array_of_dargs[1]    = MPI_DISTRIBUTE_DFLT_DARG;       /* not applicable */
array_of_psizes[0]   = 0;                              /* compute rowsize/processors */
array_of_psizes[1]   = 1;                              /* do not compute */
MPI_Dims_create( nprocs, ndims, array_of_psizes);      /* divide rows/nprocs */
MPI_Type_create_darray(nprocs, myrank, ndims,          /* define view */
                       array_of_gsizes, array_of_distribs, array_of_dargs,
                       array_of_psizes, MPI_ORDER_C, MATRIX_MPI_TYPE, &newtype);
MPI_Type_commit( &newtype );                           /* logical view handle */
MPI_Type_size( newtype, &bufcount );

MPI_File_open( MPI_COMM_WORLD, ..., &f );              /* use view on file */
MPI_File_set_view( f, 0, MATRIX_MPI_TYPE, newtype, "native", MPI_Info );
MPI_File_read_all( f, readbuf, N, MATRIX_MPI_TYPE, &status );
MPI_File_close( &f );
```

Figure 1. Logical view for a PVFS file using MPI-IO.

A logical view is defined using three (type int) arrays: *array_of_gsizes* holds the size of each dimension in the array upon which the logical view is defined; *array_of_distribs* says whether a dimension is distributed and how (NONE, BLOCK, or CYCLIC); *array_of_dargs* specifies the size of the distribution unit (BLOCK may use the default SizeOfDimension/NumberOfProcessors; CYCLIC requires a unit size); and *array_of_psizes* specifies programmer-defined number of elements to distribute to each processor if defaults are to be ignored. The view is then created and a *newtype* is defined with it. The view is then attached to a file and used.

4. FSDDS: A File System atop the Data Diffusion Space

FSDDS is a parallel file system for the Data Diffusion Space (DDS), an all-software distributed shared memory for PC Clusters. FSDDS supports typical functions to access files but also allows mapping files onto the shared address space of DDS.

4.1. DDS

DDS supports a shared address space for parallel applications running on distributed memory platforms under the SPMD (Single Program Multiple Data) model [1]. The size of the shared address space can be up to 2^{64} bytes, either on 32-bit or on 64-bit architectures. Shared data diffuses in the memory of each processor using the data, or in the disk space of each processor if need be, under a multiple-readers-single-writer protocol.

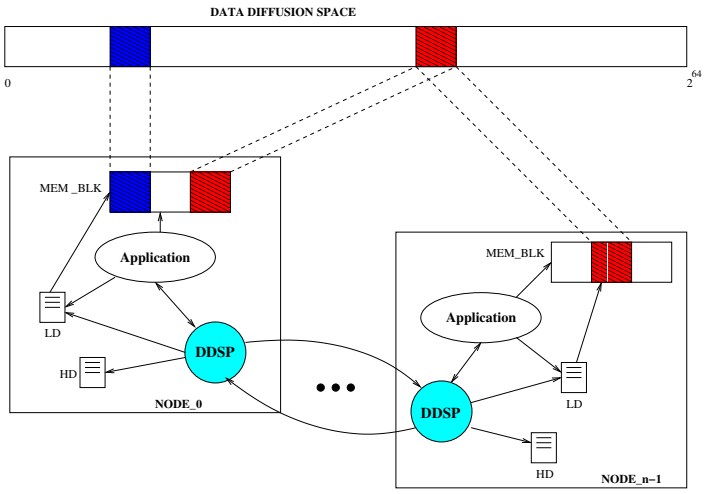


Figure 2. The DDS Architecture.

Figure 2 shows the DDS architecture. In each node runs an application process and a DDSP process. The data diffusion space is *extra* to the address space of each process running a parallel application. Shared data is dynamically mapped into the address space of whichever application process is using the data. A *local directory* is used to determine if a shared data item is already mapped (looking up its address). If it is not mapped, the sibling DDSP process sends a request to its *home directory* (HD) node, which is identified using the hash function *item-address modulo number-of-nodes*. A home directory holds the location (node) and state information (exclusive, shared, master shared) of a data item. DDSP processes communicate through TCP sockets.

4.2. File Management based on DDS

FSDDS provides applications with file management based on DDS [4]. As in other parallel file systems, in FSDDS: file data is striped across different I/O nodes, metadata is used to describe the striping, and compute nodes and I/O nodes interact as clients and servers, respectively. Data striping and metadata management are based on those of PVFS. Also as in PVFS, a node can be either a compute node or an I/O node.

When a file is opened under FSDDS, it is automatically mapped onto the shared address space of DDS. The first tera byte of DDS is reserved for shared data in arrays (i.e., not file data); the following tera bytes are used one for each file that is open. Thus files and shared data are accessed in the same way, and under the multiple-readers-single-writer protocol exerted by DDS. (The API of FSDDS is described in Section 4.3.)

For each data item in a file, its I/O node is also its home directory node; and these two roles are carried by the DDSP process in that node. However, recall that the home node of a shared (array) data item is determined with the hash function *item-address modulo number-of-nodes* , while the I/O node of a file data item is determined through metadata. Whether the hash function or metadata is used depends on the address of each data item: the address of file data items will be equal to or greater than 1 tera byte because of the mapping of files described above. This distinction also entailed some changes to the DDS replacement policy, as described below, in order to improve performance.

In each node, when the memory becomes full, a less recently used item is chosen and an action is taken depending on its status. If the status is *shared*, the item is just discarded. If it is *exclusive* and the item belongs to an array (not a file), the item is swapped onto a local temporary file. If it is *exclusive* and the item belongs to an FSDDS file, the item is sent to its I/O node, unless the *current* node is that I/O node, in which case the item is just swapped out onto its FSDDS local file. If the status of the item is *master shared* and the item belongs to an array, the item is sent to its home node, unless the current node is the home node, in which case the item is sent to another node chosen randomly. If the status of the item is *master shared* and the item belongs to an FSDDS file, the item is swapped out onto a local temporary file, or its FSDDS local file if the current node is its I/O node.

4.3. API

Figure 3 shows the addition of two matrices, $C = A + B$, using an FSDDS file for each matrix. DDS_Init is the first DDS function that must be called; it establishes communication with the sibling DDSP process, which allocates the (DDS) cache memory to store shared data, and initialises the local directory and the home directory. A file is opened/created with DDS_Open; then its data is automatically mapped onto DDS. Each processor computes $ROWS/nprocs$ rows. Before accessing data, each processor must gain access to it, through calling DDS_Write or DDS_Read. When these procedures return, the relevant data is already in the processor memory, and will remain there until the corresponding $DDS\_UnWrite$ or DDS_UnRead is issued.

Data is actually accessed through pointers held in the array dds_shmem, and the variables off_fa, off_fb and off_fc, which are associated to the file descriptors, and are *locally* shared between the DDSP process and the application process. Those variables are updated by DDSP according both to the address of the data requested with DDS_Read or DDS_Write, and to the shared address allocated to the array when it was opened.

5. Performance Evaluation

To evaluate the performance of PVFS and FSDDS we ran two applications on a 16-node PC cluster using different numbers of processors. Each node in the cluster consisted of one Intel Celeron 1.7

```
int  fa, fb, fc;                          /* Definition of file descriptors */
main(int argc, char **argv) {
  DDS_Init(NULL, NULL, mynod);            /* initializing DDS*/
  fa = DDS_Open("matrixA", O_RD | O_CREAT, NULL);
  fb = DDS_Open("matrixB", O_RD | O_CREAT, NULL);
  fc = DDS_Open("matrixC", O_RDWR | O_CREAT, NULL);

  rows = ROWS/nprocs;                     /* Each processor computes ROWS/nprocs rows. */
  offset = myid * (ROWS/nprocs);
  for (r=0; r < rows; r++){
    i = r + offset;
    DDS_Read( fa, i*COLUMNS, COLUMNS);    /* gaining access */
    DDS_Read( fb, i*COLUMNS, COLUMNS);    /* to shared data */
    DDS_Write(fc, i*COLUMNS, COLUMNS);
    for (j=0; j<NCA; j++){                /* using shared data */
      (dds_shmem[off_fc+i])[j] = (dds_shmem[off_fa+i])[j] +
                                 (dds_shmem[off_fb+i])[j] ;
    }
    DDS_UnWrite(fc, i*COLUMNS, COLUMNS);
    DDS_UnRead(fa, i*COLUMNS, COLUMNS);
    DDS_UnRead(fb, i*COLUMNS, COLUMNS);
  }                                       /* Close files */
  DDS_Close(fa);  DDS_Close(fb);  DDS_Close(fc);
  DDS_Finalize();                         /* Finalize DDS*/
}
```

Figure 3. FSDDS programming model example: matrix multiplication.

GHz processor, 512 MB RAM memory, and a hard disk. Hard disks are of different make and storage capacity (4 and 8 GB). All nodes were interconnected through a 3COM Fast Ethernet switch with 48 ports. The operating system was Linux RedHat 9.0.

The runs with PVFS used the MPI-IO interface to issue collective I/O operations. Both the runs with PVFS and the runs with FSDDS used data stored in files. In all runs, all the nodes function both as compute nodes and as I/O nodes (data is physically partitioned among all nodes). The striping of files was different for each application and is described below.

5.1. Application 1: Matrix Multiplication

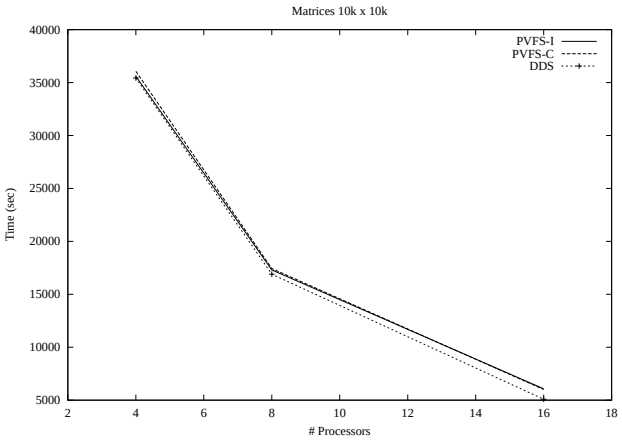
Our first application is a matrix multiplication algorithm (MM), $C = A \times B$. For $N \times N$ matrices and p processors, processor 0 computes the partial result of each element in C using the first N/p elements of the corresponding row in A and the first N/p elements of the corresponding column in B; processor 1 does the same using the second N/p elements ..., etc. Each processor computes the partial results of an entire row in C in one go, and then adds them to the actual elements in C. All processors start computing the first row in C. This algorithm matches the data access pattern with the physical data partitioning in secondary storage (by rows, according to the row-major order of the C language), and allowed us to validate the synchronisation needed to maintain data coherence on writing matrix C.

In all our experiments, each matrix is 10000×10000 long type elements (4 bytes each), is stored in a file by rows, and each file is striped across p processors (nodes). Under PVFS with *individual* (non-collective) read/write operations (PVFS-I), and under FSDDS, the stripe size in all matrices was N/p rows. We also ran a PVFS version where each processor uses collective read/write operations (PVFS-C), using a stripe of size N/p data elements for matrix A only.

Figure 4.a shows the number of read and write requests under PVFS-I, PVFS-C and FSDDS. Both PVFS versions incur the same number of read requests because they differ only in the kind of read they use: each read operation is for the same amount of data. Under FSDDS, some reads for rows in

PVFS-I/C			FSDDS	
Procs	Reads	Writes	Reads	Writes
4	12500	2500	5000	2500
8	11250	1250	2500	1250
16	10625	625	1250	625

(a) Read and write requests.



(b) Execution time.

Figure 4. MM under PVFS and FSDDS.

matrix A were satisfied from copies in other memory nodes. All versions incur the same number of write requests because they use the same write unit, a row. Under PVFS-I/C, each row is written out by one processor once it is computed. Under FSDDS, rows in matrix C are held in memory as much as possible; hence some of them are written out only when the file is closed.

Figure 4.b shows the execution time of MM under PVFS-I, PVFS-C and FSDDS. All versions show about the same execution time, even though under FSDDS less than half read requests are incurred. The reason for this is data caching. All processors access each row in matrix A starting at the first row, and once they finish computing the partial results of the corresponding row in C, they discard it; in contrast, each processor holds the rows of matrix B it uses throughout the computation. That is, each processor is accessing each row in each matrix into its memory only once, both under PVFS and under FSDDS. This implies that in PVFS some reads where satisfied from copies in main memory too, most likely from the cache of I/O nodes. FSDDS shows slightly better performance, on 8 and 16 processors, because PVFS versions used $MPI\_Gather()$ to collect all the partial results for a row in matrix C, and thus incur synchronisation cost, more so the larger the number of processors. Under FSDDS, each processor writes exclusively its partial results as they gain access to each row.

5.2. Fast Fourier Transform

Our second application applies the Fast Fourier Transform (FFT) to restore degraded or defocused images. For an image of $N \times N$ pixels, a matrix of size $N \times N \times 8$ (float type) bytes is used. From this matrix, an *images matrix* is created which corresponds to an autocorrelation process. The images matrix contains $M \times M$ images, where $M = 2N$, and is of size $2N \times 2N \times (N \times N) \times 8 = (N^4) \times 32$ bytes. To the images matrix, our application applies the FFT as follows. Processor 0 applies the FFT to the first M/p rows and to the first M/p columns (of images) along rows in each image matrix (1st), along columns in each image matrix (2nd), jumping through rows in different image matrices (3rd), and jumping through columns likewise (4th); processor 1 applies the FFT to the second M/p rows and to the second M/p columns (of images), ..., and so on. Figure 5 shows an images matrix for $N = 2$, and its partitioning for 4 processors.

We ran FFT on 4, 8 and 16 processors, both under PVFS (non-collective) and under FSDDS. In all

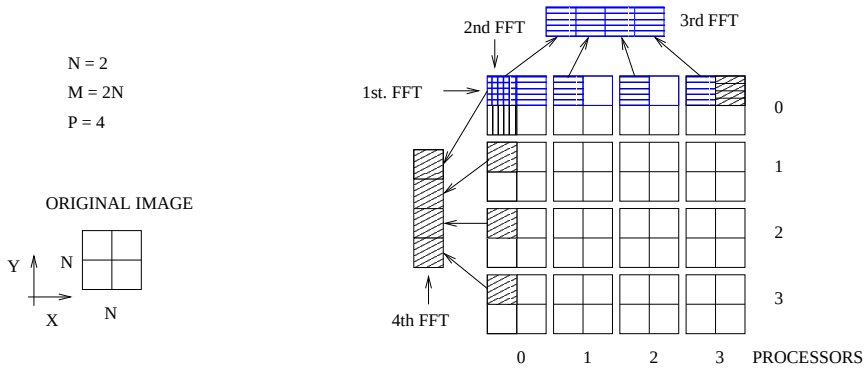


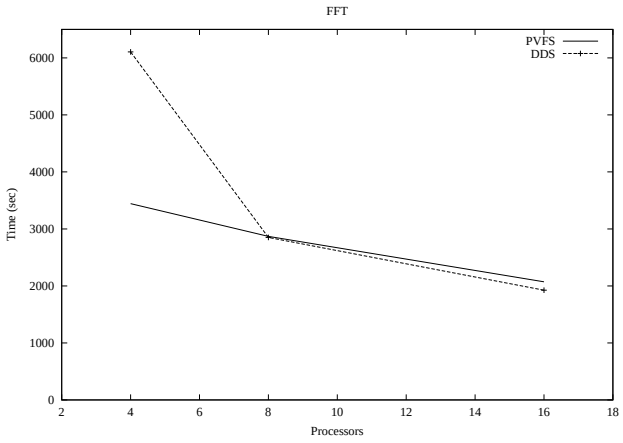
Figure 5. FFT: The images matrix for $N = 2$, and its partitioning for 4 processors.

runs, the stripe size was M/p rows of images, and the images matrix was of size $((128)^4) \times 32 = 8$ GB, and could not be held entirely in memory in all processor-count configurations.

Figure 6.a shows the number of I/O requests per processor. In each processor-count configuration, the number of I/O requests under FSDDS was smaller than under PVFS because, under FSDDS, some reads were satisfied from copies in other memory nodes, and because writes occurred in memory copies which were committed to disk only when there was a shortage of memory or until the file was closed.

	PVFS		FSDDS	
Processors	Reads	Writes	Reads	Writes
4	49152	49152	48729	48732
8	24576	24576	23638	23638
16	12288	12288	9934	9934

(a) Read and write requests.



(b) Execution time.

Figure 6. FFT under PVFS and FSDDS.

Figure 6.b shows the execution time of FFT, both under PVFS (non-collective) and FSDDS, on 4, 8 and 16 processors. On 4 processors, even though the number of reads and writes under FSDDS was smaller than the number of reads and writes under PVFS, the execution time under FSDDS was much greater because the available memory was relatively small, and thus the replacement policy of FSDDS was exerted frequently. On 8 and 16 processors, the larger memory available meant less use of the replacement policy, improving performance.

6. Conclusions and Future Work

We have outlined the use of PVFS and FSDDS, and presented some experimental results on their performance. Files in PVFS can be accessed rather simply through a UNIX-like interface; the MPI-IO interface can also be used to define logical data views. The use of views is not simple, however, even though there is some logic behind it. FSDDS interface is quite cumbersome, because DDS internal data structures are exposed. We are working on the design of an extension to the C language and its preprocessor to avoid the use of the DDS interface entirely. We envisage parallel applications will use arrays and files as a shared memory; the preprocessor will issue the corresponding DDS_Read-DDS_UnRead and DDS_Write-$DDS\_UnWrite$ pairs, and the *reference* to each shared data item based on DDS internal data structures.

The performance of PVFS and FSDDS was similar except for our second application (FFT) on 4 processors. On this application/configuration, the replacement policy of DDS had an adverse effect because the amount memory was relatively small. However, the data caching of DDS does work in general, as can be seen from the slightly better performance that FSDDS shows on 8 and 16 processors (3-10%). We are working on using different replacement policies and data striping methods to improve the performance of FSDDS.

References

[1] Jorge Buenabad-Chávez and Santiago Domínguez-Domínguez: The Data Diffusion Space for Parallel Computing in Clusters. In Proceedings of Euro-par 2005 (LNCS 3648) 61–71.

[2] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur: PVFS: A parallel file system for linux clusters. In Proceedings of the 4th Annual Linux Showcase and Conference 317–327. 2000.

[3] Peter F. Corbett and Dror G. Feitelson: The Vesta parallel file system. ACM Transactions on Computer Systems, 14 (3) 225–264. 1996.

[4] Santiago Domínguez-Domínguez, Jorge Buenabad-Chávez: Distributed Parallel File System for I/O Intensive Parallel Computing on Clusters. In Proceedings of International Conference on Electrical and Electronics Engineering and X Conference on Electrical Engineering, ICEEE/CIE2004, Acapulco, Guerrero, México, September 8-10. 2004.

[5] Jay Huber, Christopher L. Elford, Daniel A. Reed, Andrew A. Chien, and David S. Blumenthal: PPFS: A high performance portable parallel file system. In Proceedings of the 9th ACM International Conference on Supercomputing 385–394. 1995.

[6] IEEE/ANSI Std. 1003.1: Portable operating system interface (POSIX)-part1: System application program interface (API) [C Language]. 1996.

[7] Florin Isaila and Walter F. Tichy: Clusterfile: a flexible physical layout parallel file system. Concurrency and Computation, 15 (7/8) 653–679. 2003.

[8] MPI-IO: A Parallel File I/O Interface for MPI, http://www.mpi-forum.org/docs/docs.htm. 1997.

[9] Nils Nieuwejaar and David Kotz: The Galley parallel file system. In Proceedings of the 10th ACM International Conference on Supercomputing 374–381. 1996.

[10] Ron Oldfield and David Kotz: Applications of Parallel I/O. Technical Report PCS-TR98-337. Department of Computer Science, Dartmouth College, Hanover.

[11] Ron Oldfield and David Kotz: Armada: A parallel file system for computational grids. In Proceedings of the First IEEE/ACM International Symposium on Cluster Computing and the Grid 194–201. 2001.

[12] Apratim Purakayastha, Carla Schlatter Ellis, David Kotz, Nils Nieuwejaar, and Michael Best: Characterizing parallel file-access patterns on a large-scale multiprocessor. In Proceedings of the Ninth International Parallel Processing Symposium 165–172. 1995.

[13] Huseyin Simitci and Daniel Reed: A comparison of logical and physical parallel I/O patterns. The International Journal of High Performance Computing Applications, 12 (3) 364–380. 1998.

pCFS: A Parallel Cluster File System *

Paulo Afonso Lopes [a], Pedro D. Medeiros [a]

[a]CITI and Department of Informatics - Universidade Nova de Lisboa
Monte de Caparica, Portugal
email:{pal,pm}@di.fct.unl.pt

This paper proposes pCFS, a high performance shared disk cluster file system (CFS) targeted at small to medium sized clusters, which aims to support a broad spectrum of I/O intensive applications, including support for parallel I/O.

Traditional CFSs perform control operations (locking and cache coherence) over the LAN and data access over the Storage Area Network (SAN); the proposed architecture will exploit all the available interconnect infrastructures (SAN and LAN) to maximize I/O bandwidth and minimize latency. It will also offer a high level of availability without sacrificing performance by taking into account two complementary views: hardware and software. By hardware, we mean multiported disk arrays exporting RAID volumes to a SAN with multiple access paths; on the software side, pCFS will use cooperating caches with replication of modified data blocks, allowing delayed writes without the fear of data loss.

The benefits of data transfer over LAN and SAN together have already been validated by an experiment involving a modified version of the GFS cluster file system.

1. Introduction

High performance I/O for cluster architectures has been a subject for a lot of research. While several big clusters have adopted low cost-per-node distributed disk (DD) architectures where I/O nodes have inexpensive internal disks, small-to-medium sized clusters, used in scientific research and in business data infrastructures to support large data bases, have been favoring the shared disk (SD) approach.

In DD architectures compute nodes perform data movement to and from I/O nodes either across inexpensive Ethernet interconnects, or via more expensive specialized ones such as Myrinet, SCI, or Infiniband. In SD architectures, both nodes and storage devices are attached to a storage area network (SAN), an infrastructure that commonly uses Fibre Channel (FC). The cost per node for both types of infrastructures, FC and Myrinet or Infiniband, is similar.

This diversity among architectures has obviously spurred different file system approaches: distributed disk architectures are usually handled with a distributed file system (DFS), while shared disk architectures resort to cluster file systems (CFS). Some file systems are specifically designed to cater for HPC needs, and these usually have the word ”parallel” standing out in their names, e.g., *Parallel* Virtual File System (PVFS) and General *Parallel* File System (GPFS).

This paper proposes pCFS, a shared disk cluster file system which aims to achieve high performance in a broad spectrum of I/O intensive applications ranging from computational access to large data sets to video streaming and databases, including efficient support of parallel I/O.

pCFS is targeted at small to medium sized clusters where data is stored in shared devices on a SAN. It merges concepts and techniques that were successful both in established CFSs and DFSs,

*CITI is a research centre funded by the Foundation of Science and Technology of the Portuguese Ministry of Science and Universities. This research was also supported by an IBM Equinox grant.

but goes beyond current practice in the following aspects:

- while current CFSs use SANs to access storage devices and LANs for the exchange of control information, pCFS performs data access using both;
- while techniques such as cooperative caching were previously used in DFSs only to increase the size of a ”global cache”, pCFS again goes beyond its original intent, and uses it to achieve several goals simultaneously: to decrease latency, minimize data movement to and from disk devices, and increase fault tolerance.

The rest of the paper is organized as follows. Section 2 is an overview of some well known distributed and cluster file systems, and an assessment of their shortcomings. The proposal of pCFS, a new architecture that will overcome the observed limitations, and its distinctive features against both established and state-of-the-art file systems for clusters, is described in Section 3. Section 4 reports on the proof-of-concept tests carried out, and Section 5 concludes the paper with remarks on main contributions of pCFS and future work.

2. File Systems for Cluster Architectures

The implementation of a DFS or a CFS is a careful decision placement along four dimensions: a) richness and adequacy of the file model to its target architectures and applications; b) performance c) resilience, fault-tolerance and recoverability; and d) security.

2.1. Representative File Systems

PVFS [3] is well known to HPC users; it [2] uses a client/server approach, with computational nodes accessing both I/O server as well as metadata nodes through a TCP/IP network; inexpensive implementations range from Fast through Gigabit Ethernet, whereas more expensive ones use low latency, high bandwidth networks such as Myrinet or Infiniband. A PVFS filesystem is created on top of a logical group of Linux ext2 filesystems, each one stored in a local disk of a distinct I/O server; files are then round-robin stripped across these *PFVS disk units*. PVFS is integrated with the VFS interface, so existing binary applications using the standard file I/O API (memory mapped files are not supported) can be executed; it also has its own API, which implements the PVFS ”native” I/O model, and includes operations such as collective I/O. A point worth mentioning is that PVFS does not use client caching at all, at the expense of a decrease in performance; but, then, it is able to ”almost” [3] offer POSIX single node semantics without requiring the complexity of dealing with cache coherency.

On the other hand, file systems such as GFS [12] or GPFS [10] used in shared disk clusters use the SAN to transfer data, and the host interconnection network to transfer control information (e.g., locking). They are usually symmetric, [4] and are referred to as Cluster File Systems (CFS). GPFS is an IBM-proprietary shared disk file system that runs on AIX Power and on IBM supplied Linux clusters connected to an FC SAN. In a GPFS cluster, non SAN-attached nodes can still access shared data through a software layer, called Virtual Shared Disks (VSD), running on top of a general purpose network infrastructure. GPFS supports two forms of caching: client-side (CS), the standard operation mode where POSIX single-node equivalent semantics is supported, and server-side (also called data-shipping, DS). DS mode is used to boost performance in ”heavy sharing” situations, but

[2] For the purpose of this discussion the new version, PVFS2 [4], is not different from the previous one, PVFS.
[3] For a brief introduction, please read the PVFS2 User’s Guide, available on *http://www.pvfs.org/pvfs2*
[4] Every node has direct access to every storage device.

has several restrictions: it requires program modification, where processes desiring to use it issue a ”DS start” call and block other processes from accessing the file; it does not preserve POSIX single-node equivalent semantics; and it cannot be used together with some other file system calls.

GFS has been recently made into an open source CFS; a very short description of its features, without repeating ourselves, would be to say that ”it’s GPFS minus the VSD and the DS mode”.

2.2. Shortcomings of current HPC-targeted File Systems

How may these distributed file systems we have just presented, be positioned along the four dimensions listed in 2.1, thus showing their strengths as well as limitations? PVFS main functional shortcoming is the lack of support for file locking, whereas its main operational drawback is fault-tolerance: if an I/O server fails, its local disks will be unavailable, and all the PVFS file systems depending on it will fail. Possible solutions are: to use a node to store a replica of data stored in another node (e.g., using RAID ”over-the-LAN”); or to dispense with local disks altogether and instead use a SAN, with disk arrays whose LUNs are then privately mounted by the hosts. Both solutions, however, do incur in performance penalties: the software solution, because there is some amount (twice if mirroring is adopted) of overhead data being transferred to the ”mirroring host” over the network; the hardware RAID solution because, while it introduces another network, the SAN (which is expensive), it does not take full advantage of it (client nodes cannot access the disks directly using the SAN) and, worst of all, performance can be degraded due to the bottleneck that may arise if several I/O servers simultaneously access their disks to get hold of data striped on them.

GPFS, on the other hand, scores highly on all dimensions. The only aspects that we will point out are: it is not an open source product; it does not support POSIX equivalent single-node semantics in data shipment mode; and, more importantly, it does not make the ”best” use of all the available infrastructures.

There is, however, a very important issue that has been somehow neglected when championing the ”pure” DFS approach (as used by PVFS, NFS, or Lustre [1]): CPU power available for the application. It has been reported [2] that keeping a gigabit Ethernet interface running close to its full bandwidth consumes quite a lot of CPU: we have measured close to 40% of a 2.6 GHz Xeon with regular-sized, and about 30% with Jumbo frames. Thus, in small to mid-sized HPC clusters with DFSs such as PVFS or Lustre, this may be an important issue. Sure, an ”expensive” CPU offloading interface (e.g., Myrinet) can be added, but then the price advantage over SAN-based cluster file systems is lost. This is quite different from what happens in configurations with CFSs, say, GPFS or GFS, where all nodes may be compute nodes and file system clients simultaneously, and where the FC boards do offload the CPU.

3. A new, multi-purpose, Cluster File System

We will now present the driving ideas behind pCFS, including targeted environments, architecture, major building blocks and distinctive features.

3.1. Targeted hardware architectures

pCFS is targeted to shared-disk clusters; it is being designed to extract very good performance and scalability on small to medium-sized (less than a hundred nodes) clusters, where nodes are SAN--connected to several storage disk arrays. As pointed out in subsection 2.2, we have three strong arguments in favor of using a CFS on mid-sized SAN-based cluster solutions:

1. all nodes are available for running applications;

2. I/O tasks running on the nodes do not cause excessive CPU load; and,

3. the architecture is inherently fault tolerant.

3.2. Software architecture

The overall software architecture of pCFS and its relationship to other relevant kernel modules is depicted in Fig. 1 below.

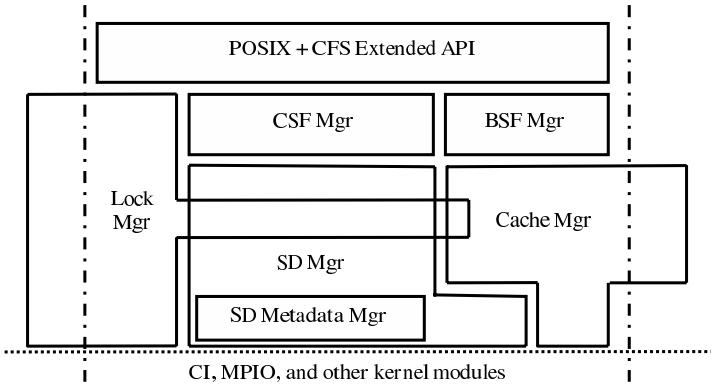


Figure 1. pCFS software architecture. pCFS modules are depicted above the dotted line and inside the ”dash-dot” lines; modules, such as Lock and Cache Manager, that extend to the outside of ”dash--dot” boundary lines represent interactions with other OS modules and subsystems; those drawn below the dotted line represent subsystems that provide services to pCFS.

The *pCFS API* can be thought as the union of three sets of primitives: standard POSIX I/O primitives, including advisory locking support; lock primitives to implement other locking semantics, such as ”Linux-style” mandatory locking; and finally a set of primitives to support parallel, strided, and collective I/O operations.

The *Shared Data* (SD) *module* is the one that ultimately performs block read/write operations to the ”physical” disks, while the SD Metadata Manager sub-module deals with the ”on disk” data structures: bitmaps, superblocks, inodes, etc., or their equivalents in the pCFS world.

The *Cache module* keeps recently accessed data blocks (or pages, depending on the grain size being used) in memory; distinct cache coherency policies may be used across distinct files according to user-defined parameters; cache coherency is implemented with the help of services provided by a lock manager. Cooperative caching [5] is a technique that has been used to extend the cache of a node with the help of memory from other nodes; if a node knows that some other, ”buddy” node, has the desired data block on its cache, it can fetch the data from that node’s memory, without the need of a disk access. Cooperative caching has been quite used in DFSs, but not in CFSs; in pCFS it has a double function:

- On reads, it may be used as a regular cooperative cache; but while in a DFS reading from another node memory is the only way to read from its local disk, in our shared disk architecture it is simply another path available to be used on reads, one that can be regarded as a powerful performance enhancing mechanism, as it enables both infrastructures, SAN and LAN, to be used for data block movement.

- On writes, the traditional cooperative cache can be extended with a form of replication to increase file system availability: instead of writing through to a disk, we can replicate the block to some other nodes and, for practical purposes, guarantee that we can flush the updated block on a node failure; or, we can take the performance enhancing route: when a node wants a modified block cached in another, we may simply forward it, thus avoiding the need to flush it to disk and re-read it from the disk in the other node.

We will need to keep track of cached blocks, and a directory-based protocol [6] or other technique usable for software coherence protocols [8] is the way we plan to do it. In brief, what we are proposing is a global unified cache across the cluster nodes.

The *Lock module* is a building block for the implementation of both user-level locking primitives (POSIX and others, as needed) and file sharing semantics on cluster-wide open files. Thus, the Lock module will be a client of some Locking Subsystem, and will have to interact with the Linux cache and VFS. Each Locking subsystem is a distinct implementation of a ”lock server” with a minimum set of operations capable to support the needs of the Locking Module; this concept, taken from the GFS ”Lock Harness” module, allows different implementations, corresponding to different levels of complexity to be used: from a simple, centralized, locking server, through a high available one, up to a Distributed Lock Manager [7] implementation.

Finally, the *Character (CSF) and Block Special-File (BFS) Managers* are the modules that provide standard character-device and block-device access to applications.

3.3. Services provided to the pCFS subsystem

The pCFS subsystem needs some services provided by other subsystems; among them are the Cluster Infrastructure (CI), Multi-path I/O (MPIO), and the I/O drivers.

Cluster Infrastructure provides Cluster Membership, a Services Database, and Reliable Communications across cluster nodes. This subsystem closely follows the ideas laid out on [11].

The Multi-path I/O module has a sub-module for each class of I/O protocol available on an interconnect; for example, every TCP/IP running device, be it an Ethernet, Myrinet, or other device is kept under a sub-module that allows that device to be used together with other devices to provide for a sort of ”link aggregation” feature that offers a wider bandwidth and higher availability path.

3.4. Distinctive features of pCFS

When comparing a full pCFS implementation with parallel distributed file systems such as PVFS, we believe the following advantages will show up:

a) existing applications that need locking will run unmodified on pCFS, while they must be modified to run on PVFS;

b) application performance in pCFS will be less dependent on data partitioning and placement decisions (although achieving the highest level of performance still requires ”optimal” striping)

c) provisions will exist on pCFS to establish QoS contracts for sustained I/O bandwidths;

d) due to its cooperative cache replication/update strategy both write and read performance can be significantly larger in pCFS;

e) maximum I/O bandwidth can be achieved in pCFS by using the whole I/O infrastructure, including both the SAN and the LAN; and,

f) there will be no single point of failure in pCFS, whereas in PVFS, if a node is down there is no way to access its locally stored data [5], and a whole file system instance will be unavailable.

Comparing the pCFS with the recently proposed Lustre file system is not an easy task; Lustre [1] is a very ambitious file system: everything is apparently covered, from security to high availability, from scalability to integration of ”legacy” file systems (such as Linux ext2, ext3, ReiserFS, etc.) with new Object Based Storage proposals, to ease of recovering after crash, etc. Unfortunately, we could find only one published report [9] and it shows Lustre performance to be similar to PVFS, and somewhat lower than GPFS. CFS Inc., is publicly releasing old versions only; the one currently available, although covering a lot of the expected functionality, seems [6] to have several annoying bugs that were fixed on releases not publicly available, and to be very difficult to install [9].

When comparing the pCFS with a cluster file system such as GPFS, we believe that the advantage expressed in e) above also applies here, while the one in d) is not quite the same: in this case, the cooperative cache will enable pCFS to achieve good performance on applications that heavily write-share the same file block across several nodes (in GPFS this is done by modifying the application to use ”data-shipping”), while still preserving POSIX single-node equivalent semantics and allowing any process to access the file using the standard API (which GPFS does not).

4. pCFS proof-of-concept tests

To validate the fundamental assumptions, we decided to make small modifications to GFS, a well established, production-level CFS, that closely implements the same architectural ideas that stemmed from the VAX Clusters research and products. After carefully evaluating Oracle’s OCFS [13] and openGFS [14] (seemingly phased out when GFS moved to ”open source” status), both documented, we ended up studying thousands of lines of GFS code (as documentation is not available) and we have opted for carrying out the tests as a simulation inside GFS kernel code.

We have modified GFS to follow one out of two different code paths when reading a file:

SAN path: When a process in one node is reading a file that is opened across the cluster, the regular GFS code path is followed: shared read locks are placed on the disk blocks, and, if another node has modified one or more file blocks, the node has to flush them out to disk before granting the lock.

LAN path: When directed to do so by the simulation test, the kernel code on the reader node follows another code path, where a) lock requests and grants are simulated by message exchange between the nodes, and b) the writer node supplies a copy of the modified page(s).

To implement the LAN path we have built two kernel modules: the client module is called by the modified GFS code when a decision has been made to get data directly from another node; it forwards the request to the other node, where the server module handles it by shipping back the data. The performance data was collected when running a single writer/multiple readers application which access the same data blocks; after producing new data, the writer signals the readers to consume it. Tests were carried out using IBM x335 dual Xeon nodes attached at 1 Gbps to a Brocade SilkWorm 2800 switch and IBM FAStT 200 storage array.

The single writer/multiple readers scenario was chosen to expose the known limitations of cluster file systems in heavy sharing situations. Figures 2 to 4 summarize the experiment results and validate the benefits of using cooperative caching as proposed by pCFS.

[5] Or we must resort to one of the solutions proposed in subsection 2.2

[6] We accessed the Lustre discussion forum on *https://lists.clusterfs.com*

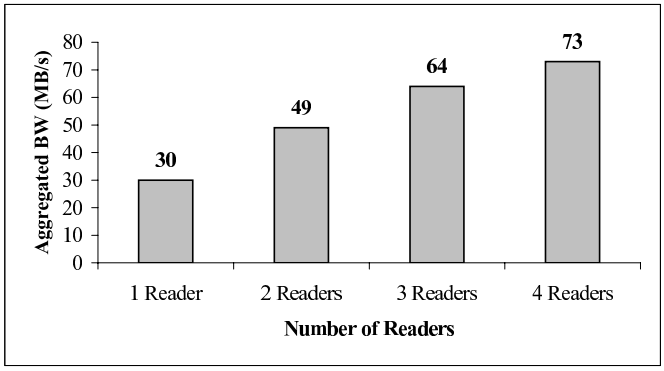


Figure 2. Aggregated application disk bandwidth in non-modified GFS. Bandwidth increases with the number of readers (one per node). Block size is 4 Kbytes.

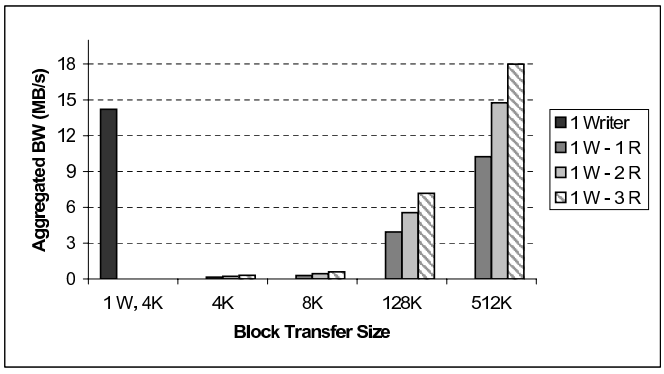


Figure 3. Single writer/multiple readers aggregated bandwidth in non-modified GFS. Block size ranges from 4 K to 512 Kbytes. Note the bandwidth drop when adding the 1st concurrent reader.

5. Conclusions

Through detailed analysis and benchmarking, we have identified shortcomings of both distributed disk and shared disk file systems. File systems for distributed disk architectures based on ”plain Ethernet” suffer from excessive CPU consumption on data movement tasks, and are usually non fault tolerant. Those for shared disk architectures are inherently fault tolerant, but do not use all the available I/O infrastructures; they also do not, as a rule, perform well under heavy sharing, and may not scale well in configurations with a large number of nodes.

We believe the proposed cluster file system architecture, by combining current ”top of the breed” hardware building blocks (FC SANs and disk arrays, together with Gigabit Ethernet, Myrinet and Infiniband) with a sophisticated software architecture that includes cooperative caches with replication of modified data blocks, will be able to maximize I/O bandwidth and minimize latency and, when compared with other file systems for distributed and shared disk architectures, offer: better performance and application compatibility (via its standard POSIX I/O API); resilience, fault tol-

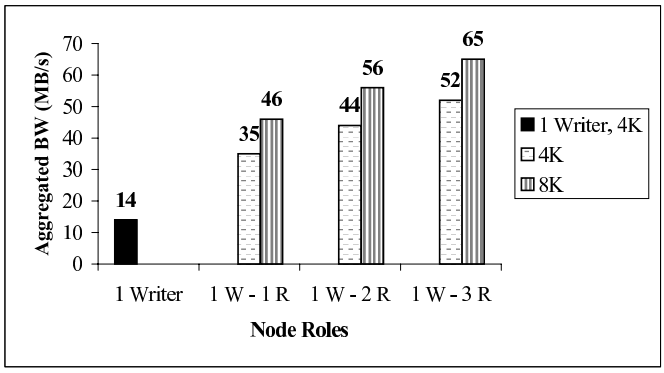


Figure 4. Same situation as figure 3, but with GFS modifications. Note that the bandwidth is much higher than in figure 3, and close to the ”readers only” experiment of figure 2.

erance and recoverability; and higher node counts than currently available CFSs. We also believe this architecture is well suited to be integrated with a SSI kernel such as Kerrighed [15], and to vastly benefit from RDMA capable interconnects. The architecture and performance of pCFS will be evaluated in a future prototype implementation.

References

[1] Braam, P. et al, The Lustre Storage Architecture, 2003.
[2] Celebioglu, O. et al, Optimizing Linux Cluster Performance by Exploring the Correlation between Application Characteristics and Gigabit Ethernet Device Parameters, Proceedings of the 5th Linux Clusters Institute (LCI) Conference, 2004.
[3] Carns,P. et al: PVFS: A Parallel File System for Linux Cluster, Proceedings of the 4th Annual Linux Showcase and Conference, 2000
[4] P. Carns, Achieving Scalability in Parallel File Systems, PhD Thesis Clemson University, 2004.
[5] Dahlin, M. et al, Cooperative Caching: Using Remote Client Memory to Improve File System Performance. Proceedings of the First Symposium on Operating System Design and Implementation, 1994
[6] IEEE Standard No.: 1596, Scalable Coherent Interface, 2000
[7] Kronenberg, N. et al. VAXclusters: A Closely-Coupled Distributed System. ACM Transactions on Computer Systems, Vol. 4,No. 2, 1986
[8] Li, K. and Hudak, P. , Memory Coherence in Shared Virtual Memory Systems. ACM Transactions on Computer Systems, 7(4):321-359, 1989
[9] Margo M. et al, An Analysis of State-of-the-Art Parallel File System for Linux, 5th LCI (Linux Clusters Institute) Conference, 2004
[10] Schmuck, F. and Haskin, R., GPFS: A Shared-Disk File System for Large Computing Clusters, Proceedings of the Conference on File and Storage Technologies (FAST’02), 2002
[11] Cluster Infrastructure for Linux. http://ci-linux.sourceforge.net
[12] GFS Project Page, http://sources.redhat.com/cluster/gfs/
[13] OCFS Project Page, http://oss.oracle.com/projects/ocfs/
[14] openGFS Project Page. http://opengfs.sourceforge.net/
[15] Vallé, G. et al. A Case for Single System Image Cluster Operating Systems: Kerrighed Approach. Parallel Processing Letters, 13(2), 2003.

Parallel I/O optimization for an air pollution model

David E. Singh[a], Félix García[a], Jesús Carretero[a]

[a]Departamento de Informática, Universidad Carlos III de Madrid, 28911 Leganés, Spain

This work presents and evaluates different I/O parallelization approaches for the Sulphur Transport Eurelian Model 2 (STEM-II), a large-scale pollution modelling application that is used to simulate air quality conditions. STEM-II is a computationally intensive application that requires of a multiprocessor environment for performing simulations in a reasonable response time. Due to the large amount of data that uses, the I/O becomes a critical factor for the application performance. This paper is focused in the study and optimization of this stage for distributed memory systems. Several parallelization approaches are presented and evaluated for a Cluster of PCs. Experimental results show that the efficient parallelization of the I/O achieves a significant reduction in the overall execution time.

1. Introduction

Nowadays, the air pollution related to high populated or industrial areas is a topic of increasingly social interest. In particular, it is especially useful the use of simulation tools for providing feedback mechanisms that allow limiting the pollutant levels. STEM-II [1] is an air quality model that simulates transport, chemical transformations, emission and deposition processes in an integrated framework. This model was successfully used for the control of the emissions of pollutants produced by the Endesa power plant of As Pontes (Spain). In addition, the STEM-II was chosen as case of study in the European CrossGrid project, proving its relevance for the scientific community for its industrial interest as well as its suitability for the high performance computing.

In terms of application performance, STEM-II is a computationally intensive application that requires a multiprocessor environment for performing simulations in a reasonable response time. In [2] several parallelization approaches were presented proving that STEM-II can be efficiently executed on a multiprocessor environment. However, the parallelization of the I/O stage was not performed, being an important bottleneck for the application performance. This topic is studied in detail in this work and efficient parallel I/O techniques are also presented. Additionally, we present two main contributions: First, an improved parallel implementation of STEM-II is shown, where the I/O output stage is completely parallelized. Second, a comparative study of different I/O parallel strategies is performed, evaluating the impact in the application performance of several parameters such as the locality degree, problem size or number of processors. This study allows introducing feedback mechanism than can be of interest to develop optimized I/O techniques.

In [3,4] the I/O requirements of several parallel applications are analyzed, showing that access patterns to non-contiguous small volumes of data are frequent. FLASH code [5] is broadly used as I/O benchmark given that it represents a real application where the access pattern is non-contiguous both in memory and in file. A similar I/O behaviour is also exhibited by the parallel implementation of STEM-II studied in this paper.

The paper organization is as follows: Section 2 describes the air quality model STEM-II, presenting its internal structure as well as its parallel implementation. The parallelization of the I/O stage is studied in Section 3. Specifically, several I/O parallel techniques are presented and discussed. The performance is analyzed in Section 4 and Section 5 summarizes the main conclusions of this work.

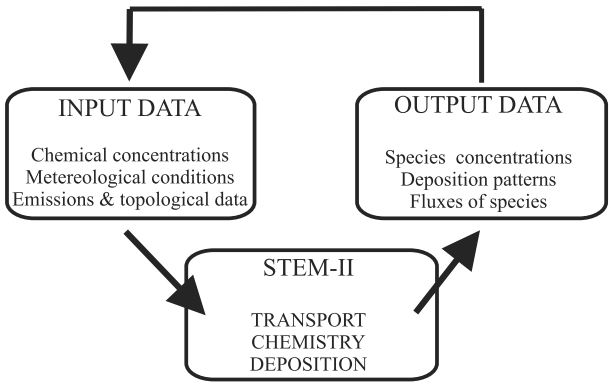


Figure 1. General block diagram of STEM-II.

2. STEM-II air quality model

Endesa power produces a power of 1400 MW obtained from the combustion of coal. This coal is a mixture made up of local lignite (with a high content in sulphur) and other foreign coals (with low sulphur concentrations). In this context, the problem of knowing the optimal mixture appears so that the maximum local lignite yield is obtained while satisfying the limits of emission of pollutants. STEM-II is used to predict the atmospheric pollution considering the weather forecast.

STEM-II is a 3D grid-based model that simulate $SO_x/NO_x/RHC$ multiphase chemistry, long-range transport and dry+wet acid deposition. This application is used for calculating the distribution of pollutants in the atmosphere from specified emission sources such as cities, power plans or forest fires under particular meteorological scenarios. The prediction of the atmospheric pollutants behaviour includes the simulation of a large set of phenomena, including diffusion, chemical transformations, advection, emission and deposition processes. A detailed mathematical description of the physical and chemical mechanisms included in this model can be found in [1].

In terms of code structure, STEM-II has a modular design that permits to split the computations of each simulated process. Figure 1 shows a simplified block diagram of the application. It consists of three major functional blocks: A simulation module, an input data module for loading the environment conditions and an output data module for exporting the simulation results. The former module includes all the computational-intensive elements required by the simulation and comprises the STEM-II kernel. More specifically, this kernel consists of a transport stage, which computes the emission and transport of the pollutants; a chemistry stage, which simulates the transformation of chemical species by gas + liquid phase chemistry and a deposition stage, which describes the dry and wet deposition of pollutants. The input data module acquires the initial chemical concentrations, topological information and meteorological data such as temperature, humidity levels or precipitation rates. Finally, the output data module exports the gaseous and aqueous concentrations of each modelled specie, as well as other relevant simulation results like reaction rates or the amount of deposited elements.

These three functional blocks are within a temporal loop that specifies the duration of the simulation. In our experiments, each time step corresponds to a minute of real time where the kernel is executed. The I/O blocks are executed for a pre-defined interval of iterations, sixty in our case.

The main motivation for developing a parallel version of this application is that in the case of Endesa power plant, STEM-II requires of meteorological input data that are available less than 12h

```
DO  x = 1, n_x
      DO  y = 1, n_y
            vertcl routine
               DO  z = 1, n_z
                        ...
               END DO
            asmm routine
               DO  z = 1, n_z
                        ...
               END DO
            rxn routine
               DO  z = 1, n_z
                        ...
               END DO
      END DO
END DO
```

Figure 2. Pseudocode of vertlq routine

before the simulation. Once those data are available, a 12h interval exists for the calculation of a 2-day prediction. In this way, it is possible to determine if future emissions will be within the limits imposed by the European norm, or if it is necessary to change the coal mixture to reduce emissions. For the sequential application, this time is too short for the computational requirements of the model, being necessary its execution on a parallel system.

This work is focused on the parallel STEM-II implementation for distributed memory systems. In [2] a detailed study of the code was performed, proving that the $vertlq$ routine (member of the chemistry module) is the most costly part of the application. The organization of this routine is shown in Figure 2. Note that it presents a nested structure, with calls to other subroutines. Dependence analysis proves that the accesses throughout the dimensions x and y do not produce data dependences. This is not the case of the rest of the dimensions[1] where different kinds of dependences appear. Based on this study, an implementation of the STEM-II for distributed memory system was performed, parallelizing both x and y loops. In [2] a detailed study of the application performance for different degrees of loop parallelism was achieved. Results show that the most efficient parallel implementation is the one that executes x loop sequentially and y loop in parallel. This behavior can be explained if we take into account the Fortran style of array storage by columns. When the x loop is executed sequentially, each processor follows the same order that the data present in memory, accessing to the data by columns with a higher degree of locality.

Figure 3 shows the structure of the parallel version, coded in Fortran 77 and using the MPI standard library for implementing communications. The root process sequentially loads and initializes the application data. Then, matrices are distributed among the processors by means a $scatter$ operation, allowing to execute in parallel the $vertlq$ routine. Part of this data are subsequently gathered to proceed to the I/O output stage and another part (that does not have to be the same) is also gath-

[1]For reasons of space only z loops are shown. There are internal nested structures where different species and phases are also traversed.

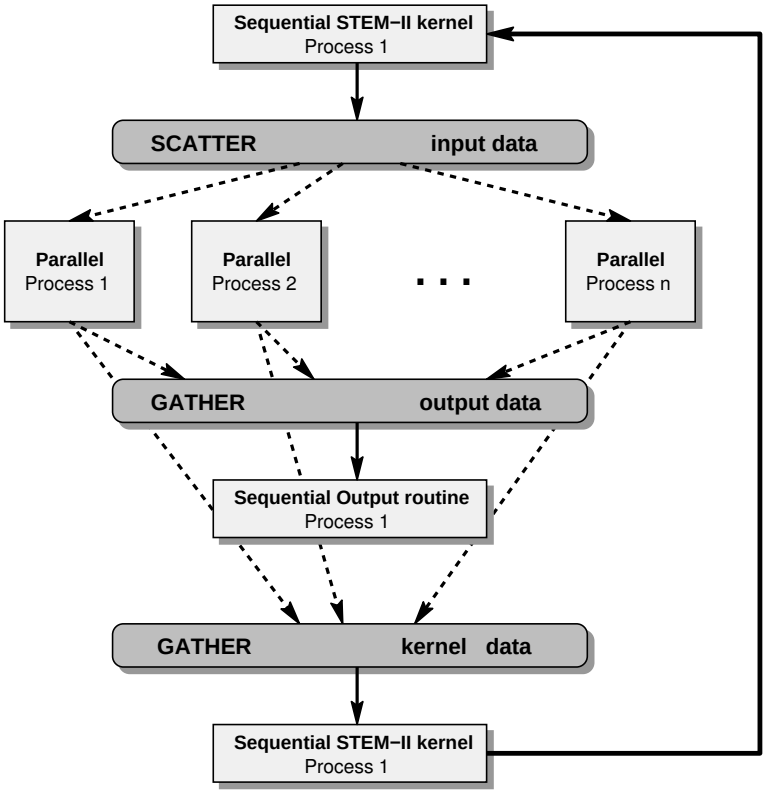


Figure 3. Diagram of the parallel implementation of STEM-II .

ered to compute the sequential version of kernel. Next section presents different alternatives for the parallelization of the I/O stage that avoid the use of gather communications.

3. STEM-II I/O parallelization

STEM-II produces a huge volume of data related to the gas-phase concentration and deposition patterns for each instant of time. In a multiprocessor environment, there are two reasons for storing part of these data in disk: the first one is the use of monitorization tools for analyzing the state of the simulation; the second reason is the use of fault-tolerance mechanisms that allow restoring a previous execution state. Before starting the simulation, the user selects the variables to be stored in disk and the frequency (in number of iterations) in which the data are exported. In both situations, the high volume of data stored to disk causes a significantly increase of the I/O cost.

Regarding the I/O input stage, there are two main reasons for not including it in our study. Firstly, the cost of this stage is small, given that the amount of loaded data are reduced. Secondly, this stage can be executed in parallel using a shared filesystem (for example Network File System) because it just includes read-only operations that do not produce access conflicts.

In this paper we propose the use of MPI I/O operations implemented in ROMIO Library [6] for performing parallel disk storage. Two techniques, called *interval* and *block* disk access, were developed and tested using PVFS distributed filesystem [7]. In contrast, their performance was compared with two local-filesystem oriented techniques called *sequential* and *local* disk access. All of them are based on the data distribution used by the parallel implementation of the STEM-II.

Now, we summarize each one of them.

- **Sequential disk access** represents the original write operation used by the parallel version of STEM-II. Figure 4(a) shows an example of this scheme. Initially, the data are distributed using a block-cyclic distribution. With a collective MPI_GATHERV communication, the root processor reconstructs the complete array. Subsequently, this processor sequentially stores the data on its private filesystem.

- **Local disk access** represents an alternative for exploiting private filesystems on a multiprocessor execution environment. In this approach (see Figure 4(b)) each processor writes its local data on its private filesystem. Given that no communication operation is required and only private filesystems are used, this operation can be performed in parallel with a high efficiency. However, it presents two main disadvantages: the original data format is not kept and the filesystems are replicated (not shared), so the global data are not accessible for the processors (including the root).

- **Interval disk access** corresponds to a parallel I/O operation on a distributed filesystem. Figure 5(a) shows the structure of this operation. Initially, each processor creates a MPI datatype related to its local distribution. This datatype includes both the entries that fit in one block of data[2] and the gap of entries that conforms the stride. Then, a MPI_FILE_SET_VIEW operation is applied to the filesystem, so that each processor only sees its assigned portion of the logical file. Finally, a MPI_FILE_WRITE operation is applied, forcing all the processors to write on different parts of the filesystem. Note that in Figure 5 we distinguish the logical view of the filesystem from the physical layout. In PVFS files are distributed using a round-robin scheme. In the figure, we assume that the stripping file size is the half of the memory block size. Interval disk access approach keeps the original data format at expense of a low locality in disk accesses. In addition, the global data are accessible for all processors.

- **Block disk access** performs a parallel I/O operation with a greater locality degree. Like interval disk access, each processor creates a datatype that represents its assigned block of data. However, the information is stored in consecutive entries of file by means of a MPI_FILE_WRITE_AT operation. Figure 5(b) illustrates an example of this technique. Note that the file layout is different from previous proposals (sequential and interval disk access) although all the stored data are still globally accessible for all processors.

In the next section the performance of these proposals is compared and the impact of effects like locality degree or filesystem architecture are analyzed.

4. Performance analysis

As test platform we used a cluster of PCs consisting of eight compute nodes interconnected by a GigaEthernet at 1000 Mb/sec. Each node consists of two Intel Pentium III at 1GHz, with 1GB of memory and 200GB of disk. The operating system is a Linux Debian 2.4.19 and the Fortran compiler is GNU f77 version 2.95.4 using optimization level -O3 and linked (if necessary) with MPICH1 (v1.2.4) libraries. We have used PVFS (v1.6.3) in our implementation of distributed filesystem with a striping factor of 64KB. Private filesystem corresponds to local $ext3$ partition of Linux.

[2]Note that the datatype can be different for each processor given that the block size can change.

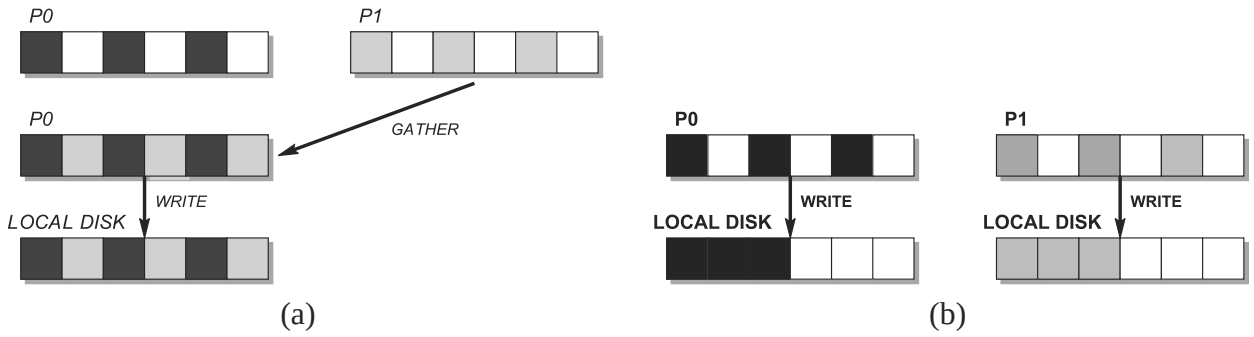


Figure 4. Examples I/O techniques on local filesystem (a) sequential disk access, (b) local disk access.

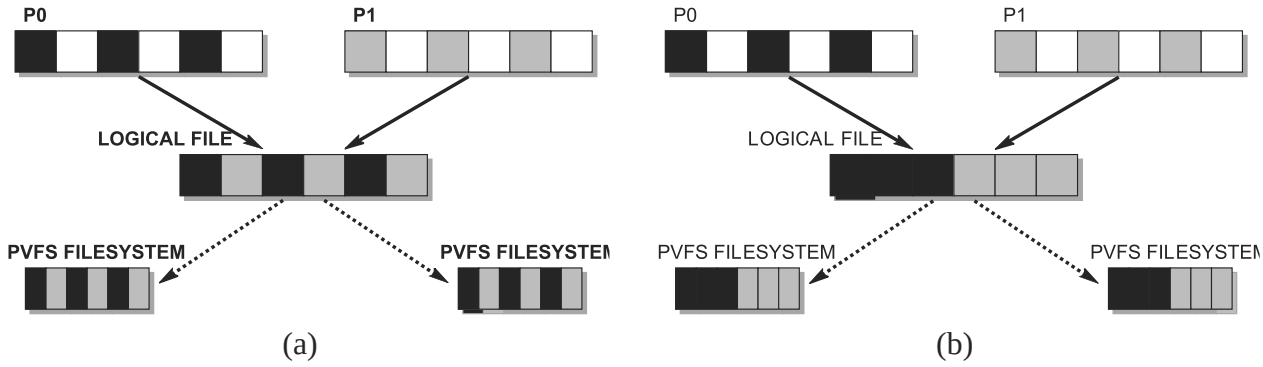


Figure 5. Examples I/O techniques on distributed filesystem (a) interval disk access, (b) block disk access.

In terms of application performance, the considered environment covers an area of 61 x 61 km^2 centred in As Pontes power plant with a resolution of 1 km^2 and 15 height levels. For this scenario different meshes are used, ranking from 61x61 (2D surface) to 61x61x15x56x3 (3D mesh extended with two dimensions related to 56 chemical species and 3 phase states). In this context, we can conclude that STEM-II is a highly expensive application in terms of amount of consumed resources, especially CPU and memory.

In this work two different sets of data were considered for being stored on disk: *Set1* consists of three 4D arrays and two 3D arrays that store the specie concentration in gas phase, summarizing 26MB of data. *Set2* adds one 5D array, up to 62MB of data. This set contains the chemical distribution of all the species considered in the simulation.

Figure 6(a) compares, for *set1*, the execution time of the initial code (called *un-optimized*) with the *interval* and *block* techniques. The un-optimized time includes the cost of the communications plus the execution time of the write operation which has a constant value of 0.19 secs. Note that for the un-optimized code the communication cost strongly increases with the number of processors. This is due to the fact that the data distribution exhibits a poor locality when a gather operation is applied. This effect is more important when the number of processors increases.

Figure 6(b) shows equivalent results for *set2*. In this case, the sequential execution time of the write operation associated to the un-optimized version has a constant value of 0.44 secs.

Several conclusions can be extracted from these graphics: Firstly, the sequential access to disk

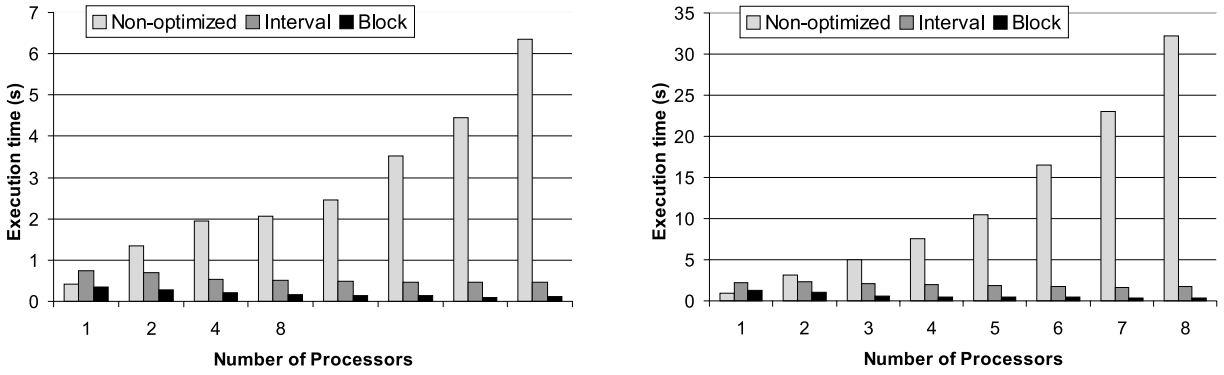


Figure 6. Execution time of un-optimized, interval and block techniques for: (a) $set1$ data, (b) $set2$ data.

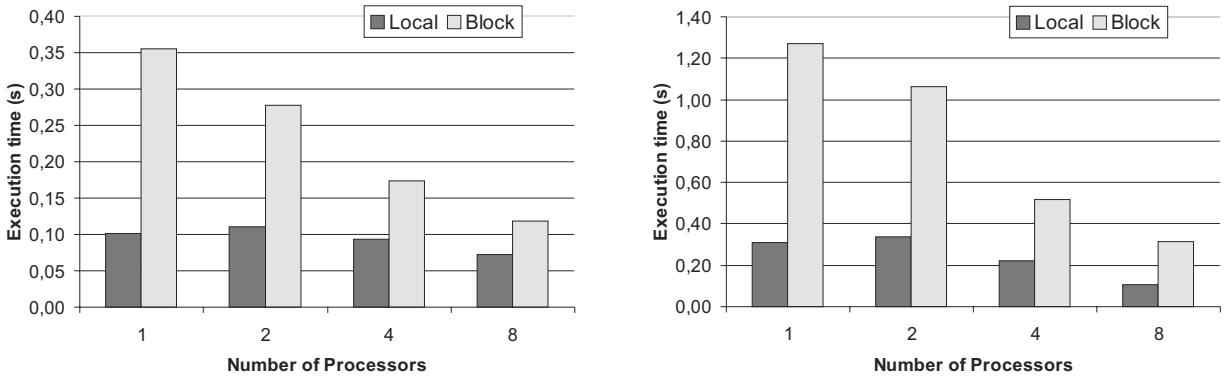


Figure 7. Execution time of local and block techniques for: (a) $set1$ data, (b) $set2$ data.

(implemented in the un-optimized version) is very inefficient due to the cost of the communications. For example, sequentially storing the contents of *set1* each iteration, represents the 30% of the overall execution time for a two processor execution. Secondly, with our proposal the I/O performance increases when the PVFS filesystem is used. Interval disk access technique implies an extra overhead respect to the sequential which causes a poor efficiency for executions with one processor. This overhead is related to the use of a distributed filesystem (instead of a local filesystem of the un-optimized version). The overhead decreases when the number of processors increases, proving the correct sustainability of the proposal.

Due to the increase of data locality, block disk access technique obtains a better scalability for all considered scenarios at spite of different file layout. Figure 7 compares the performance of this proposal with local disk access technique. We remark that the first one uses the PVFS filesystem, offering a globally accessible file whereas the last one uses a local filesystem in which only portions of the problem are stored (which makes its use impractical in most of the cases). Local disk access technique is the most efficient solution, but the performance of block technique strongly increases

as the number of processors grows. In this situation the performance tend to be the same in both proposals.

We have tested our proposals using PVFS2 (v1.1.0) distributed filesystem with both MPICH1 (v1.2.6) and MPICH2 (v1.0.2) library. Despite evaluating different optimization levels in both (MPICH and PVFS2) configurations, the performance obtained was inferior to PVFS1 filesystem, thus, the measurements were not included in this work.

5. Conclussions

In this work several parallel I/O techniques were introduced for increasing the parallelism degree of the STEM-II application. Additionally, a comparative study of local and distributed filesystem performance was presented. Results show that with the use of parallel I/O the application performance can be considerably increased. The reduction in the number of communications and the parallel disk access are the main advantages of the use of parallel I/O techniques. In addition, experimental results show that the locality degree in disk access has an important impact in the I/O performance. In case of being possible to allow transformations in the file layout, the block disk access technique is the most adequate. As future work, we propose to improve two-phase I/O techniques where the disk locality could be increased keeping small communication costs.

Acknowledgements: This work has been partially supported by the Spanish Ministry of Science and Education under the TIN2004-02156 contract.

References

[1] Carmichael, G.R., Peters, L.K., Saylor, R.D. *The STEM-II regional scale acid deposition and photochemical oxidant model - I. An overview of model development and applications.* Atmospheric Environment, 25A, 10, pp. 2077-2090, 1991.

[2] María J. Martín, David E. Singh, J. Carlos Mouriño, Francisco F. Rivera, Ramón Doallo, Javier D. Bruguera. *High performance air pollution modeling for a power plant environment.* Parallel Computing, 29, pp. 11-12, 2003.

[3] Phyllis E. Crandall, Ruth A. Aydt, Andrew A. Chien and Daniel A. Reed. *Input/Output Characteristics of Scalable Parallel Applications.* In Proceedings of Supercomputing '95, 1995.

[4] Nils Nieuwejaar, David Kotz, Apratim Purakayastha, Carla Schlatter Ellis and Michael Best. *File-Access Characteristics of Parallel Scientific Workloads.* IEEE Transactions on Parallel and Distributed Systems, 7(10), pp. 1075-1089, 1996.

[5] Fryxell B., Olson, K., Ricker P., Timmes F. X., Zingale M., Lamb D. Q., MacNeice P., Rosner R., Truran J. W. and Tufo H. *FLASH: An Adaptive Mesh Hydrodynamics Code for Modeling Astrophysical Thermonuclear Flashes.* Astrophysical Journal Supplement, 131, pp. 273-334, 2000.

[6] Rajeev Thakur, Ewing Lusk and William Gropp. *Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation* Technical Memorandum ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory, 2004.

[7] P. Carns, W. Ligon III, R. Ross and R. Thakur. *PVFS: A Parallel File System for Linux Clusters*. In Proceedings of the Third Annual Linux Showcase and Conference, pp. 317-327, 2000.

A Parallel Data Management Layer for Data Mining

M. Coppola[a], P. Pesciullesi[b], L. Presti[c], R. Ravazzolo[b], M. Vanneschi[b]

[a]Ist. di Scienza e Tecnologie dell'Informazione, CNR - Via Moruzzi 1, 56124 Pisa, Italy

[b]University of Pisa, Dip. di Informatica - Via Buonarroti 2, 56127 Pisa, Italy

[c]IMT Lucca Institute for Advanced Studies - Via San Micheletto 3, 55100 Lucca, Italy

We propose the design of a data management abstraction level to implement a full set of parallel KDD applications, with minimal performance overhead and greater scalability than conventional DBMS, providing a high-level parallel API to be exploited by parallel and out-of-core data mining algorithms. Our approach exploits knowledge of the parallel and sequential structure of applications. Programs are developed with the ASSIST parallel programming environment, and expose explicit algorithmic hints in the sequential code through the data management API. We describe an existing prototype and report examples and first test results with mining algorithms.

1. Introduction

The design of the data management level for Knowledge Discovery in Databases (KDD) involves several difficult trade-offs in choosing the right API. There are contrasting needs in the implementation of this layer w.r.t. expressive power, flexibility, raw performance (e.g. I/O performance, computational overhead), whose balance conditions the overall performance of the KDD process.

Because of its iterative and interactive search nature, KDD highly benefits from the use of standard DBMS tools, exploiting their flexibility in the steps of data extraction and preparation. During the Data Mining (DM) phase, on the other hand, size of data and number of attributes often rule out algorithms with high accuracy, just because their complexity in terms of in-core and out-of-core operations [12] makes them impractical in real-life situations. In order to minimize this effect, data management support for Data Mining must achieve high efficiency and performance. Different solutions have been used in the practice, ranging from flat-file access, the development of special-purpose API to conventional DBMS [8], to RAM-based DB and OLAP approaches [9].

When developing parallel KDD systems we are confronted with even more complex issues, as larger and harder problem instances have to be solved, while efficiently exploiting parallel I/O, and memory hierarchies made up of several stacked sequential and parallel architectural layers. These issues are not addressed by conventional DBMSs, which fail to scale up to massively parallel architectures.

We propose the design of an intermediate abstraction level, the Parallel Data Repository (PDR), that provides enough flexibility to implement a full set of KDD applications, exposes performance critical choices to the application programmer hiding the messy details, and can be implemented with high performance on parallel architectures. We want to avoid

- the high overhead that standard DBMSs impose in order to support relational views, atomic transactions and concurrent access at the record level,
- the drawbacks of sequential/parallel low-level approaches to programming, which are complex, error prone and seldom portable,

- the limits in scalability that conventional parallel approaches incur, being based on shared-memory DBMS servers and/or database replication.

We define an abstract software architecture for this data management support, which aims at exploiting out-of-core techniques from within structured parallel programs, reducing the implementation complexity of parallel data mining applications without sacrificing their performance. The intended use of such a software layer is to ease parallel processing of data in the mining and validation steps of the KDD process. We thus assume that clean input data can be exported from conventional DBMS or Data Mart to the high-performance management system, in order to execute parallel mining algorithms on it.

The design of the PDR has been partially implemented, and it has been tested with several algorithms [3], showing promising results.

In Sec. 2 we discuss the approach we have taken in designing the system, and compare it with previous work. Sec. 3 gives details about the PDR structure, and Sec. 4 about those parts that have already been implemented. Sec. 5 shows simple examples of Mining algorithms that can be efficiently expressed using the API of the PDR and presents preliminary benchmark results. Sec. 6 outlines future work directions.

2. Approach and Related Work

When applying parallel and distributed computing techniques to KDD systems and algorithms, we need to decompose data access and computation workload in parallel. Our aim is to reach *architecture portability* of DM computational cores, to be able to extend and reuse them as KDD modules or as the basis of different DM algorithms, without sacrificing their performance and parallel scalability. We define an abstraction level for data management, in order to optimize communication performance and workload distribution and, at the same time, to grant sufficient expressive power to code the algorithms independently from the details of data access. The approach we pursue has four key features

1. efficient parallel modularity/decomposability of computations exploiting the interface,
2. block-oriented, efficient exploitation of memory hierarchies,
3. DM tailored data management implementation and data semantics,
4. low overhead with respect to raw I/O.

The first point is addressed by exploiting the ASSIST structured parallel programming environment [1,11] for writing DM algorithms [4]. The parallel coordination approach allows to clearly express the parallel behaviour of the application, while sequential code performing the work doesn't deal with the issues of concurrent access to data. By decoupling the local (to each block or partition of the data) and global parts of the computation into different modules, we can control the flow of data in the algorithm structure itself, thus also avoiding the need for access control in the data management layer. In our view this requires

- independent concurrent operation on partitions of a file (a feature that is not provided by plain POSIX, but is needed for parallel I/O [10]),
- support for user-defined synopsis data structures linked to data blocks; each process/module in the DM application should be able to efficiently build/update/fetch the sufficient statistics needed to dispatch a data block within the algorithm and/or to a different processing node.

The structured approach to parallelism is coupled with a block-oriented data management level. We exploit the common structure of many DM algorithms, which are mainly data intensive, and can be written to work as much as possible on large blocks of data, improving

the I/O performance. The theory of out-of-core (OOC) algorithms has already shown that explicit secondary memory access control is needed to achieve optimal results. State-of-the-art libraries for OOC programming like TPIE [2] are based on the load/unload paradigm. Block selection is specified by the user algorithm and is implemented by a block-moving engine, performing all I/O and related optimizations.

We concentrated on the parallel aspect of OOC, thus we do not have yet developed an API and an engine for prefetching strategies like that of TPIE. Recent works exist on the combination of the OOC and parallel aspects [5] into the FG framework. FG allows to easily organize block (pre)fetching and parallel load-balancing for programs essentially structured as pipelines, while preserving the modularity of the sequential code performing the computation on each block. With respect to FG, ASSIST programs are not restricted to the pipeline pattern[1] and can be run over the Grid as well as on cluster platforms. On the other hand we are currently not going to integrate the expressive tools for OOC and parallelism computation in the ASSIST coordination language. Through the PDR design we support out-of-core parallel operations on a memory hierarchy by providing a block-oriented interface to the processes of a parallel application, which can then exploit block-aware algorithms to maximize the amount of in-memory computation.

Assuming that the data has been cleaned and consolidated into a single large table, efficient support of parallel and secondary-memory block-oriented operation for DM algorithm is much easier to achieve than in the general case of DBMS applications, as we can focus on the problem of handling large bi-dimensional matrices with fixed row schema. For the sake of performance, we assume that the in-memory data representation is the same as those in other memory levels, in order to avoid conversion overheads and to directly access tables loaded from secondary memory.

We thus advocate an intermediate approach between using DBMS tools and flat files, on the one hand providing only basic operations, with low computational cost, on the data tables. On the other hand, we improve w.r.t. relational databases and to flat-files in the ability to express data types routinely used in mining algorithms. Support of those basic data types used in DM algorithms that have a compact and efficient machine encoding, like small integers, sets of labels and booleans, also addresses the requirement of low overhead I/O.

Our approach differs from the prevalent one of developing special purpose API to conventional or parallel DBMS (Microsoft OLE-DB is an example [8]). The scalability of such an approach is limited by that of (most often SMP-based) parallel DBMS. When aiming at high performance DM and on-line transaction processing, RAM-based approaches are also used (e.g. the GemStone OLAP solution based on a distributed cache approach [9]).

3. Design Proposal

We propose a distributed SW architecture where sequential, parallel/concurrent clients can access the PDR. The PDR is structured as a multiple-layer memory hierarchy sketched in Fig. 2, where we have three levels: the memory local to each process (M0), a shared-memory primary level (M1) and a secondary memory level (M2).

Global memory is thus distinguished into primary and secondary. The secondary level is implemented through the secondary memory of a set of computing nodes.

If we employ the current technology of computational clusters, and a hardware or software

[1]Generic graphs of parallel modules and explicit loops can be expressed and controlled in ASSIST.

implementation of the primary shared memory level, it is safe to assume that primary global memory (M1) is smaller than secondary one (level M2, which is the aggregation of the available discs) and at least two orders of magnitude faster w.r.t. access latency. With the same assumption, access to local discs, when they are present, and to non-local disk storage, exhibits the same latency. The PDR however does not exploit the three-level memory hierarchy as such. Levels M0 and M2 are used to process/hold data, while memory level M1 is reserved to hold block-related meta-data, i.e. additional information about each block.

We show in Fig. 1 the layout of data within the repository. We call each database managed in the PDR a dataset. All records in a dataset have the same structure, defined in term of provided types (integers, floats, dates, boolean, raw data, record keys) and of user-defined nominal types (defined as sets of labels). Thus the dataset is a bi-dimensional table with heterogeneous columns.

Blocks are the smallest amount of data transfer and of parallel work decomposition. Their size is fixed at dataset creation: a larger block size typically increases I/O bandwidth and DM algorithm efficiency, and decreases the available parallel work on a dataset. The external memory paradigm is applied to manage the data, using levels M0 and M2.

The upper level of the PDR provides the API and implements all local data management functions. The user code interfaces to the data by means of C++ classes, that manage data buffers in main memory, allow to operate on meta-data and delegate the I/O to the lower implementation levels.

Dataset meta-data represent the fixed schema of the dataset rows, including definitions of user-defined types. The schema is managed by the PDR and is kept linked with each dataset, with low I/O overhead and memory occupation. It doesn't need to live in shared memory, as it is fixed at dataset creation[2].

Each block of data is also a linked to a data space in the M1 level (Fig. 1), where synopsis data structures defined by the program exploiting the PDR are kept. The purpose of this additional space is to speed up application execution: programs can quickly store and retrieve synthetic information about a data block, to choose which blocks to process next in sequential/parallel computations, and sufficient statistics, useful to DM algorithms to avoid loading the data at all. We show a few examples in Sec. 5.

Synopsis data structures will have a separate API from that of the data (allowing to define and manipulate them) and a different implementation. We rely on the assumptions that sufficient statistics are much smaller than the dataset, can change according to the algorithms, have a dynamic structure, and need to be shared among different parts of the algorithms much more often than the large blocks of the dataset. Thus we conclude that sufficient statistics should be stored on a fast memory that also supports synchronizations, like the (virtually) shared memory level M1.

I/O of data blocks is implemented by a lower level of parallel data servers with minimal centralized support to coordinate them. A block transfer engine can be implemented on each separate computing node, cooperating with the I/O servers.

The PDR design is architecture-independent. However, we want to avoid any data conversion across the memory levels, thus we assume an homogeneous architecture, with the same kind of CPU and O.S. to execute all processes, and a common runtime (C++) to access

[2]Meta-data management can be a bottleneck for PVFS. In our case PVFS handles only a few, very large files, and we do not support arbitrary changes in the dataset structure. Hence we don't incur in performance losses caused by the PVFS meta-data server, and PDR's own global meta-data are read-only.

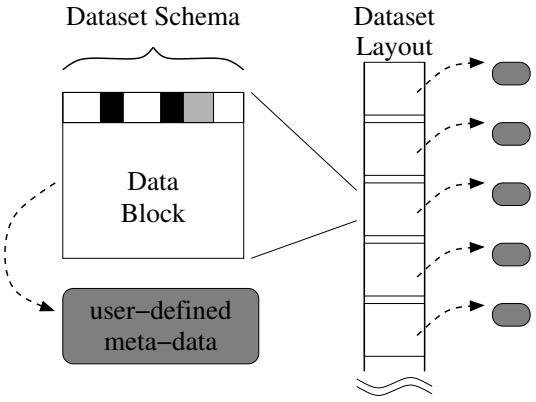


Figure 1. Layout of data within the PDR.

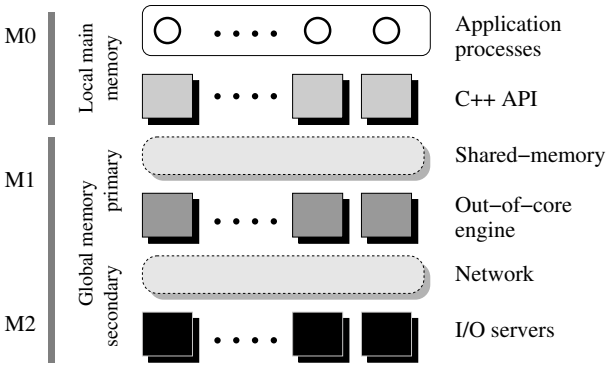


Figure 2. Implementation layers of the PDR.

in-memory data.

4. Current Implementation

We carried on the first implementation of the Parallel Data Repository as part of a parallel KDD engine within the SAIB project [3].

Mining Algorithms in the SAIB system are structured parallel applications written with the ASSIST parallel programming environment. They are developed following a common set of interfaces and used as interchangeable basic components within the KDD system. The PDR is used as an external object (an active object interfaced to all application processes and possibly being itself implemented as a parallel application). We exploit the parallel structure of the applications to ensure that parallel activities in the algorithm do actually operate on separate subsets of the data blocks.

The prototype PDR is structured as two layers, an *interface* level and an *implementation* one. The interface layer is implemented by a shared library, interfacing application processes to the data. At this level each dataset has a logical *schema* defined at creation, and can have multiple *views*. A view is a subset, possibly reordered, of the main schema of a dataset. It can be used to access the data, allowing algorithms to ignore and be independent of the actual structure of a dataset.

The interface layer provides high-level functions to (1) manage datasets life cycle (2) define logical schemes of datasets and dataset views, (3) load and unload blocks of data from multiple datasets to in-memory tables, (4) access to, and management of in-memory tables, also performing integrity checks. The implementation layer will operate on a *physical schema* that is usually different form the dataset main logical schema. The distinction between them, and the API to manipulate logical schemes, exist because programs can choose to operate only on part of the attributes of a dataset, and they also hide record field reordering which is performed by the PDR implementation to compact the data layout.

The interface layer performs also read/write operations of data blocks from shared/parallel devices. With respect to the abstract architecture of Fig. 2, the implementation of the interface layer merges within the C++ API the essential functions of the OOC engine.

The PDR implementation layer performs out-of-core data block transfer from secondary memory, exploiting different I/O supports (POSIX I/O, parallel file system). We have employed a parallel file system (PVFS version 1 [7]) to implement the I/O layer shown in Fig. 2.

I/O servers actually map transparently to the *iod* PVFS daemons. This choice was convenient for the first implementation as it avoided the immediate need for developing a parallel OOC engine. We also support sequential file-systems, with NFS as a special case. It is thus possible to share a PDR dataset with sequential applications, loosing the parallel I/O advantages, but exploiting the same API. Combining multiple disc spaces into a single PDR space, effectively reimplementing the parallel file system functionalities, is something possible with respect to the abstract architecture we envisioned, but as a research direction it is yet unexplored.

The shared-memory level M1 has not yet been integrated in the PDR architecture. We emulate its functionalities exploiting in the algorithm the shared memory data-structures provided by the ASSIST run-time. Shared dynamic data structures can be defined and used within the program. The prototype has the full functionality of the abstract architecture, except for the ability to define synopsis data structures that are automatically persistent with the dataset, and for the need to explicitly manage these structures in the program code.

As a final remark, the proposed architecture can also be exploited on large clusters to pursue a RAM-based approach. In the general case, however, to hold medium size databases in (virtual) shared memory changes the parameters of the memory hierarchy exploited, and makes the proposed PDR architecture less useful.

5. Examples

Several algorithms from the data mining field can exploit the PDR interface, and most clustering algorithms fall in this category [6]. The BIRCH and CURE approaches are based on sufficient statistics and representatives. The STING grid-based approach and the density-based approach of OPTICS and DBSCAN rely on parallel and secondary memory techniques in order to optimize the running time. Many parallel and sequential optimizations for the classical k-means/medoid clustering algorithms are based on multiple levels of summary information associated with spatial data partitions.

Many low-level tasks (e.g. sorting, searching) can be expressed using the PDR primitives, as they allow to emulate those offered by OOC frameworks like FG [5].

Classification by tree induction is another notable source of examples: these algorithms are divide and conquer in nature, and recursively split the input dataset to build the classification tree. At each split, we need to compute a set of statistics over the data associated to the current tree node (histograms of the combinations of different attribute values within each data partition).

We have developed a prototype of the C4.5 algorithm that interfaces to the PDR. The structure of the prototype is described in Fig. 3. It is composed of two interconnected ASSIST modules, one of them exploiting both data and task parallelism in different phases of the computation. Blocks are assigned by a control module to processing elements in the parallel module, either to exploit data-parallel computation of the statistics on a single node of the classification tree, or to perform a task-parallel expansion of separate nodes. The program reorders the dataset as needed to keep node-related partitions into separate sets of blocks. Each block, or set of blocks, is represented within the algorithm by its linked meta-data, which include the array holding histograms of values for the data in that block. Actual data transfer from the Data Repository happens only when needed to perform in-memory computation on the block.

Meta-data associated to data-blocks are used in another example, where an iterative clustering algorithm [3] needs to establish how many unclustered records belong to each block.

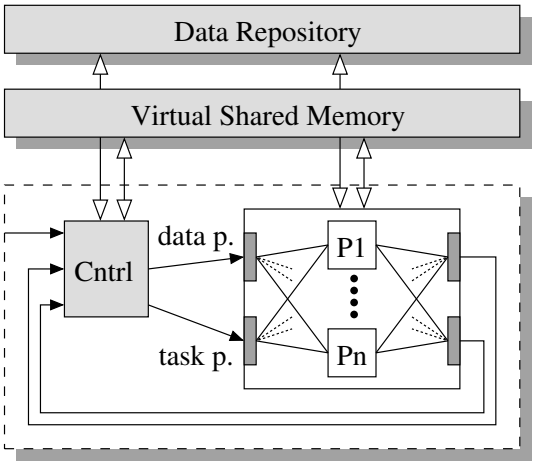


Figure 3. High-level structure of the C45 prototype.

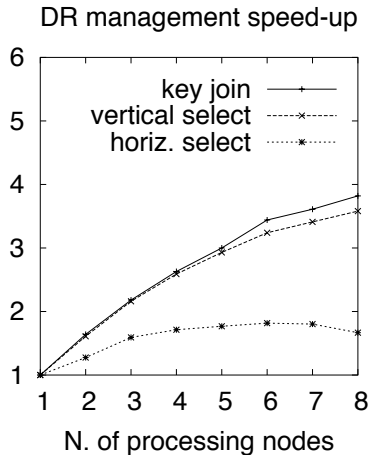


Figure 4. Performance benchmarks of the PDR implementation.

In that case we can check in constant time if a data rearrangement operation is worth on the whole dataset, to improve the execution time of the following linear scan phases.

We have also developed algorithms for simpler data-management tasks like sorting, filtering and merging of datasets. As a general remark on them, algorithms based on the scan pattern point out the need of a block prefetch API. Fig. 4 reports test results on a cluster of 8 Pentium-4 processing nodes linked by Gbit Ethernet. We used a varying number of parallel processing nodes, while keeping fixed to 8 the number of PVFS I/O nodes. We show, for a 2GB input file, the speed-up of a horizontal selection (splitting in two datasets the records of a dataset, according to a simple predicate), a vertical selection (splitting dataset records in two datasets with half of the columns) and a key-join operation (reconstructing the original file from the vertical split results).

6. Conclusions

We have introduced an architecture for a Parallel Data Repository to be coupled with high-level, structured parallel languages in the implementation of Data Mining algorithms. The PDR is based on the exploitation of the out-of-core programming paradigm and on a semantic tailored to mining algorithms, which are data-intensive and often employ a simple bi-dimensional view of the input data.

We are now working to verify that the PDR provides the right level of expressiveness to implement parallel DM algorithms with minimal coding effort, ensuring portability and high performance. This is the same goal that our research group more generally pursues w.r.t. parallel and Grid programming, and that frameworks like FG [5] aim at w.r.t. out-of-core, cluster-based computing.

The current status of development already shows some of the advantages of the architecture: we were able to develop a modular, parallel KDD engine based on the PDR, which runs on a cluster and interfaces to a parallel file system. Current block transfer engine is quite simple, though. While we already interface to parallel I/O resources, we do not yet offer the ability to do intensive data prefetch under algorithm control.

To fulfill the design of the PDR requires us to further develop the prototype, by providing

an explicit link between each data block and the corresponding synopsis data structures, as well as an API to define and operate automatically on them, in such a way that the associated data and their update code can be permanently attached to a dataset when needed. Another development we are currently evaluating is to perform some basic data-reduction and data-parallel operation in the I/O server, to reduce further network bandwidth requirements for I/O, and to ease the design of very simple parallel data management algorithms.

Acknowledgments

This work has been supported by the SAIB Project on High-performance infrastructures for financial applications, funded by MIUR and leaded by ATOS Origin, by the Italian MIUR FIRB Grid.it project, n. RBNE01KNFP, on High-performance Grid platforms and tools, and by the Italian MIUR Strategic Project L.449/97-2000 on High-performance distributed enabling platforms.

References

[1] M. Aldinucci, M. Coppola, M. Danelutto, M. Vanneschi, and C. Zoccolo. ASSIST as a research framework for high-performance Grid programming environments. In J. C. Cunha and O. F. Rana, editors, *Grid Computing: Software environments and Tools*. Springer, January 2006. (ISBN: 1-85233-998-5, to appear, draft available as TR-04-09, Dept. of Computer Science, University of Pisa, Italy, Feb. 2004).

[2] Lars Arge, Rakesh Barve, David Hutchinson, Octavian Procopinc, Laura Toma, Darren Erik Vengroff, and Rajiv Wickeremesinghe. *TPIE User Manual and Reference*, 0.9.01b edition, November 1999. Draft.

[3] Massimo Coppola, Paolo Pesciullesi, Roberto Ravazzolo, and Corrado Zoccolo. A Parallel Knowledge Discovery System for Customer Profiling. In Marco Danelutto, Domenico Laforenza, and Marco Vanneschi, editors, *Euro-Par'04 Parallel Processing*, number 3149 in Lecture Notes in Computer Science, pages 381–390, 2004. ISBN: 3-540-22924-8.

[4] Massimo Coppola and Marco Vanneschi. High-Performance Data Mining with Skeleton-based Structured Parallel Programming. *Parallel Computing, special issue on Parallel Data Intensive Computing*, 28(5):793–813, 2002. ISSN: 0167-8191.

[5] Thomas H. Cormen and Elena Riccio Davidson. Fg: A framework generator for hiding latency in parallel programs running on clusters. In *17th International Conference on Parallel and Distributed Computing Systems (PDCS 2004)*, pages 137–144, 2004.

[6] Jiawei Han and Micheline Kamber. *Data Mining, Concepts and Techniques*. Morgan Kaufmann, 2001.

[7] W. B. Ligon III and R. B. Ross. PVFS: Parallel virtual file system. In Thomas Sterling, editor, *Beowulf Cluster Computing with Linux*, pages 391–430. MIT Press, 2001.

[8] OLE-DB technical documentation. `http://www.microsoft.com/data/oledb`.

[9] GemStone Systems. Gemfire Enterprise Technical WhitePaper. `http://www.gemstone.com/products/gemfire/GemFireTechnical_WP.pdf`, 2005.

[10] Rajeev Thakur, William Gropp, and Edwing Lusk. A Case for Using MPI's Derived Datatypes to Improve I/O Performance. In *Proc. of SC98: High Performance Networking and Computing*. IEEE, November 1998.

[11] Marco Vanneschi. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing*, 28(12):1709–1732, 2002.

[12] Jeffrey Scott Vitter. External Memory Algorithms and Data Structures: Dealing with MASSIVE DATA. *ACM Computing Surveys*, 33(2):209–271, June 2001.

Compiler Techniques

Reducing Cache Misses by Loop Reordering

E. Herruzo[a], G. Bandera[b], E.L. Zapata[b], O. Plata[b]

[a]Department of Electronics, University of Córdoba, Spain

[b]Department of Computer Architecture, University of Málaga, Spain

This paper presents a novel method to determine the best loop reordering of a perfect loop nest with the aim of maximizing the resulting cache set occupation. The method is based on a simplified analytical model of the cache, reducing the cache behaviour of interest by a small number of parameters. These parameters contain the meaning of the probability of self-interference in the cache due to memory references to a particular array. Based in such parameters, we have designed a fast and effective algorithm to reorder (permute) the loops in the nest so that self-interferences in cache due to references to a particular array is minimized. An evaluation of the method is also presented.

1. Introduction

During the last decades speed of processors has been widely improved, however memory speed could not keep pace. Memory hierarchies, and specifically cache memories, were introduced to solve the performance penalty related to the speed gap. Memory hierarchy introduces the latency problem to access data. The memory latency has been attacked from two different fronts. On the one hand, by means of hardware solutions, like lockup-free caches, prefetching, out-of-order execution, and so on. On the other hand, compiler techniques have been developed to fully make use of the available hardware structures. The efficiency of architectural improvements depends on the ability of the compiler to change the structure of programs for taking full advantage of them. The most important compiler optimizations are basically loop and data layout transformations.

We present in this paper a new method to determine the best loop permutation of a perfect loop nest which tries to maximize the cache set occupation. There are several algorithms for loop permutation in the literature [2,6,9,11], being the *Loopcost* algorithm [4,1] one of the best known. Our loop permutation method is based on a simplified analytical cache model, resulting in a small set of parameters to determine the cache behaviour we are interested in. *Loopcost* shares some similar basic principles, but our method has significant differences both in the implementation and in the cache model, so as it provides a better optimization of the cache use for many important cases.

The rest of the paper is organized as follows. Section 2 presents the simplified analytical cache model. The algorithm we propose for loop permutation is introduced in Section 3. Section 4 presents some related work, and Section 5 shows the experimental evaluation of our method, compared to *Loopcost* (and a commercial compiler). Finally, section 6 draws some conclusions.

2. Modelling the Cache Behaviour

We present in this section a simplified model of the cache behaviour when specific data access patterns to memory originate from the execution of a loop nest. The aim is to define a number of parameters that characterize such a cache behaviour and could be used to determine the most suitable loop reordering, so as cache set occupation is maximized.

In order to state the model, we will consider a $k-$way set-associative cache, of size $C \times L \times W$ (C is the number of cache sets, L is the number of blocks per set, and W is the block size in words).

We also consider a $M-$dimensional array X, stored in a column-major order, which is referenced within a perfectly nested loop.

Without loss of generality, we assume that array references are made inside a $M-$depth nested loop with iteration vector $\vec{I} = (I_1, I_2, \ldots, I_M)$. Expressions in the array dimensions are of the form $f_k * I_k$, $k = 1, \ldots, M$, but any general affine expression is perfectly valid.

When the multidimensional array X is allocated in memory, it is linearized and laid out in some order. We can obtain the reference in memory to X for each iteration of the loop nest by means of a LMAD (Linear Memory Access Descriptor) [14]. So, the memory reference due to the access to X in some iteration $\vec{I} = (I_1, I_2, \ldots, I_M)$ of the nested loop is as shown (a column-major order is assumed),

$$MemRef(X, \vec{I}) = f_1 * I_1 + \ldots + f_k * I_k * \prod_{i=1}^{k-1} D_i + \ldots + f_M * I_M * \prod_{i=1}^{M-1} D_i, \quad (1)$$

where D_i is the size of the $i-$dimension of X.

The *stride* of array X on loop index I_k is defined as the distance in memory of array entries referenced by consecutive iterations of loop k, that is,

$$Stride(X, I_k) = f_k * I_k^{l+1} * \prod_{i=1}^{k-1} D_i - f_k * I_k^l * \prod_{i=1}^{k-1} D_i, \quad (2)$$

where I_k^l represents the $l-$th iteration of the loop k.

To simplify the explanation, we restrict the analysis of the cache behaviour to a single array X inside a perfectly nested loop. However this analysis can be easily extended to several arrays appearing in the same loop and to not-perfectly nested loops. The goal of our analysis is to describe and represent array self-interferences in cache due to the execution of the loop nest. That is, to determine how memory blocks with data from X are located in the cache and may replace other previously placed blocks with data from the same array.

As a first step of our study we must define a number of cache parameters. They will be used afterwards to define the proposed reordering method. During the execution of the loop, blocks of data from main memory are located in the cache. These blocks are mapped to the cache sets as shown in the following equation,

$$Set(X, \vec{I}) = \Big\lfloor \frac{MemRef(X, \vec{I})}{W} \Big\rfloor \bmod C. \quad (3)$$

The execution of consecutive iterations of loop k generates memory references separated a distance of $Stride(X, I_k)$. The distance (in cache sets) of the blocks containing these referenced data, once they are placed in cache, is a some sort of a cache *set stride*, defined as,

$$SetStride(X, I_k) = \frac{Stride(X, I_k)}{W}. \quad (4)$$

There is no module operation in the above expression because we are considering only the linear placement of cache blocks (from the beginning to the end of the cache). From this point on, the rest of blocks are placed starting again from the beginning of the cache and may produce self-replacement (that is, replacement of blocks containing X data previously placed). And this process

may be repeated several times (until all iterations of loop k are exhausted). Note also that during this first linear placement of cache blocks, each block is allocated in a different cache set.

Now, we can calculate the total number of memory blocks that can be placed in the cache with no opportunity of self-replacement (by dividing the total number of sets in the cache by the cache *set stride*). After that, the rest of the iterations of loop k may produce self-replacement. On the other hand, we can easily calculate the total number of memory blocks occupied by all the array entries of X that are referenced during the whole execution of loop k. All these blocks must be placed in the cache. Dividing this value by the number of blocks that fits the cache in a linear placement, we obtain a parameter which approximates the number of linear block placements in the cache due to references to X during the execution of the loop k. This parameter is denoted by $CacheTurns(X, I_k)$ and is calculated as follows,

$$CacheTurns(X, I_k) = \frac{N_k * Stride(X, I_k) * SetStride(X, I_k)}{C * W}, \tag{5}$$

where N_k is the number of iterations of loop k. $CacheTurns(X, I_k)$ depends on the stride in memory of array X in loop k, the number of iterations of such loop, and the cache properties. It can be seen that if $CacheTurns(X, I_k)$ is less or equal to one, then all the references to array X in loop k fit the cache with no self-replacements. Otherwise, if such parameter is greater than 1, a probability of self-replacement exists. In fact, higher values of it represent higher opportunities of self-replacement.

The $CacheTurns()$ parameter gives some rough information about the miss behaviour of the cache for some specific memory access patterns. We show in this paper that this information is enough to decide how to arrange the loops in the nest in order to reduce the probability of cache misses due to self-interferences.

In order to obtain a simple reordering algorithm, we disregarded two important issues that, however, do not influence much the effectiveness of our method. On the one hand, the fact of not taking into account the possible interference with other arrays used in the same loop nest. On the other hand, the fact that there is not always a self-replacement on the array elements already contained in the cache (set associativity).

3. Loop Permutation

We present in this section an algorithm to decide the loop arrangement (permutation) that minimizes the cache miss rate due to array self-interferences. As explained previously, a high value of $CacheTurns()$ implies a high probability of cache block replacement due to self-interference. Taking this fact into account, our permutation algorithm looks for loops obtaining higher values of $CacheTurns()$ and places them in the outermost position of the loop nest. As a result, loops with small values of $CacheTurns()$ will be placed in the innermost positions. This way, cache blocks are re-used before having the opportunity to be replaced by self-interference. Fig. 1 outlines the permutation algorithm, that we call *TCacheTurns*.

In contrast to other authors, our permutation algorithm relates not only the number of loop iterations to the characteristic parameters of the cache but it also takes into account how the data inside the nested loop is referenced (stride). Note also that our algorithm agrees with the fact that, in general, it is better to have the stride-1 array references in the innermost loop of the nest in order to exploit spatial locality [3,1,4].

for (each array X) **do**
  for (each loop I_k) **do**
    Compute $Stride(X, I_k)$ and $SetStride(X, I_k)$.
    if (I_k is in $MemRef(X, I)$) **then** Compute $CacheTurns(X, I_k)$.
    else $CacheTurns(X, I_k) = 0$.
  end for
end for
for (each I_k) **do**
  for (each array X) **do**
    Add $CacheTurns(X, I_k)$ to $TotalCacheTurns(I_k)$.
  end for
endfor
Place loops in the nest in decreasing order of $TotalCacheTurns(I_k)$.

Figure 1. Loop permutation algorithm (*TCacheTurns*)

4. Related Work

Some authors attempt to unify loop and data layout transformations. Kandemir et al. [10] present a method that uses ILP (Integer Linear Programming) to calculate optimal solutions for data transformations. Their paper describes an approach to detect the data layout of different arrays in memory, together with the best loop permutation for each loop nest in the source code. Although they include an iterative method to calculate memory layout transformations, they do not indicate any precise algorithm for loop permutation. We have found other works describing an heuristic solution to both data layout and loop transformations [13], but neither of them present algorithms to determine the code transformation.

Ghosh et al. [8] describe an approach to calculate the Cache Miss Equations (CME). Their algorithm uses the reference reuse vector, instead of using the stride. One of the main problems of this approach is the expensive process to calculate the miss equations. D'Alberto et al. [7] present a static analysis of parameterized loop nests. It is based on CMEs and on static cache parameters. They use this analysis to detect the interferences or cache block replacements of the same/distinct array references.

The work presented by Clauss and Meister [5] is mainly focused on spatial locality optimization. Their method consists in providing a new array reference function to the compiler. This is a parameterized cost function based on polytopes and Ehrhart polynomials from the iteration space of a loop nest. Clauss et al. [6] and Loechner et al. [12] use the same framework to define optimization techniques for the TLB. In this paper, we also work with the polyhedron defined by the iteration space of a loop nest. We need to know the data access layout (or polyhedron structure) to determine the cache memory occupation for each loop.

The Data Relation Vectors are defined by Kandemir and Ramanujam [11] to describe a framework to establish some compiler optimizations to improve data reuse. Weikle et al. [15] use these vectors and a mathematical model of the cache to establish a formal notation that enables compiler optimizations.

Carr et al. [4] define the *loopcost* algorithm, that is used by Allen and Kennedy [1] to develop

Table 1
L2 data cache misses for different loop counts (multiplication of 800×800 matrices)

M	N	P	Algorithm	Loop Arrangement	L2 Misses
400	400	400	Loopcost	j, k, i	6925
			TCacheTurns	j, k, i	6925
400	800	800	Loopcost	j, k, i	33910
			TCacheTurns	j, k, i	33910
400	400	800	Loopcost	j, k, i	28870
			TCacheTurns	k, j, i	16180
400	800	400	Loopcost	j, k, i	9570
			TCacheTurns	j, k, i	9570
800	400	800	Loopcost	j, k, i	110520
			TCacheTurns	k, j, i	24270
800	400	400	Loopcost	j, k, i	11210
			TCacheTurns	j, k, i	11210
800	400	200	Loopcost	j, k, i	5610
			TCacheTurns	j, k, i	5610
800	200	400	Loopcost	j, k, i	8070
			TCacheTurns	k, j, i	6510

a loop permutation technique which, in a particular way, is similar to the one described here but with some important differences. Usually the loop permutation resulted from both algorithms is the same, but we obtain a different one when the array is multi-dimensional and there are several loop indices in the non-contiguous dimension. The result is also different when the loops in the nest have different sizes. The main difference between both algorithms lies in how is calculated the cost for every reference within the innermost loop. The goal of the *loopcost* algorithm is to set the loop that occupies less cache sets as the innermost loop. The *loopcost* approach classifies array references into groups that exhibit temporal or spatial reuse. In our case, we take into account the stride defined by the LMAD of every reference, and we only join references when they exhibit spatial-group reuse. In the section about experimental results we compare both algorithms.

5. Experimental Results

This section evaluates our permutation algorithm *TCacheTurns* and compare it with *loopcost* and a real commercial compiler. Our experiments were conducted in a platform with a R10k processor running in an exclusive mode. We have used the MIPSPro Fortran90 compiler (version 7.30) with $-O0$ compiler optimization option. The hardware counters of the processor have been tested using *Perfex functions*. We carried out two sets of experiments, comparing L2 data cache misses. The first set of tests is based in the matrix-matrix multiplication problem, while the second one compares the different solutions for several benchmarks extracted from NAS, PerfectB and SPEC2000.

5.1. The matrix-matrix multiplication problem

We accomplished two different types of tests: the first one analyzes the effect in L2 cache miss rate of different loop counts in the matrix multiplication problem with fixed-size input matrices (all three matrices are of size 800×800). The second one changes the size of the input matrices but

Table 2
L2 data cache misses for different matrix dimensions (loop counts $M = N = P = 1000$)

X matrix dim.	Y matrix dim.	Z matrix dim.	Algorithm	Loop Arrangement	L2 Misses
1000,1000	1000,1000	1000,1000	Loopcost	j, k, i	1.2 Mill
			TCacheTurns	j, k, i	1.2 Mill
1000,1000	1000,1000	3000,3000	Loopcost	j, k, i	1.3 Mill
			TCacheTurns	j, k, i	1.3 Mill
1000,1000	3000,3000	1000,1000	Loopcost	j, k, i	1.5 Mill
			TCacheTurns	j, k, i	1.5 Mill
1000,1000	3000,3000	3000,3000	Loopcost	j, k, i	1.6 Mill
			TCacheTurns	j, k, i	1.6 Mill
3000,3000	1000,1000	1000,1000	Loopcost	j, k, i	9.6 Mill
			TCacheTurns	k, j, i	4.2 Mill
3000,3000	1000,1000	3000,3000	Loopcost	j, k, i	9.6 Mill
			TCacheTurns	j, k, i	9.6 Mill
3000,3000	3000,3000	1000,1000	Loopcost	j, k, i	9.7 Mill
			TCacheTurns	j, k, i	9.7 Mill

keeping the loop counts. Table 1 shows the experimental results for the first type of tests. There are three cases where the loop arrangement resulted from *loopcost* and *TCacheTurns* algorithms are different. That occurs when the number of iterations of loop k is much larger than that of loop j. In all cases our algorithm obtains a lower number of L2 data cache misses.

Table 2 shows a comparison between *loopcost* and *TCacheTurns* for the second type of tests. All loops are of the same size, a total of 1000 iterations each one. This set of experiments shows that normally the loop arrangement and the number of L2 cache misses resulting from applying both algorithms is the same. However, our approach obtains a different loop reordering with a lower number of cache misses for some cases. This occurs when the size of the first matrix (X) is greater than the size of the other two. In this situation, as the k loop iterates over the non-contiguous dimension of the X matrix, the corresponding stride ($Stride(X, k)$) will be greater than for the other two matrices Y and Z. The size of X is large enough to produce a different loop arrangement and, consequently, a better performance.

5.2. NAS, PerfectB and SPEC2000 benchmarks

A comparison of the different considered solutions using a selection of various benchmarks is shown in Table 3. It can be seen that, for multidimensional arrays, our algorithm outperforms the other two approaches. In the rest of cases, the number of L2 cache misses obtained is the same. The reason for this behaviour comes from the use of the stride parameter in the determination of the best loop permutation. *Loopcost* only uses the number of iterations and considers the same cost for loops traversing non-contiguous dimensions. However, this different way of compute the best loop arrangement does not make any difference for 2D arrays with the same number of iterations, as it is the case of `fftpde` and `swim`. Some cases in which *TCacheTurns* obtains a better loop permutation than *loopcost* occur when the nest contains loops accessing non-contiguous dimensions of multi-dimensional arrays. Together with the above two algorithms we include in this table the results obtained with the native compiler with the loop permutation option switched on ($-LNO$).

Table 3
Benchmark subroutines, running conditions and L2 data cache misses

PerfectB B.	Subrout.	Loop counts	Array dim.	Algorithm	Loop Arrangem.	L2 Misses
adm	hyd	100,100,100	100,100,100	Loopcost	j,k,i	64.680
				TCacheTurns	k,j,i	19.260
				Compiler	compiler selection	64.680
flo52	collc	200,200,200	200,200,200	Loopcost	j,n,i	351.660
				TCacheTurns	n,j,i	3.450
				Compiler	compiler selection	351.660
dyfesm	mnlbyx	500,500,500	500,500	Loopcost	j,n,i	203.524
				TCacheTurns	n,j,i	37.080
				Compiler	compiler selection	162.668
migration	migrat	200,200,200	200,200,2,200	Loopcost	j,k,i	358.920
				TCacheTurns	k,j,i	4.100
				Compiler	compiler selection	358.920
NAS Benchm.						
appbt	l2norm	48,48,48,5	5,50,50,50	Loopcost	i,j,k,m	2.550
				TCacheTurns	k,j,i,m	34
				Compiler	compiler selection	2.560
appsp	spentax3	30,30,600	660,33,33	Loopcost	j,k,i	89.850
				TCacheTurns	k,j,i	3.340
				Compiler	compiler selection	89.230
fftpde	transx	500,500	1000,1000	Loopcost	i,j	4.500
				TCacheTurns	i,j	4.500
				Compiler	compiler selection	4.500
SPEC2000 B.						
gangel	polnel	80,80,80	100,100,100	Loopcost	j,l,i	158.270
				TCacheTurns	l,j,i	73.970
				Compiler	compiler selection	158.270
applu	rhs	45,45,45,5	5,50,50,50	Loopcost	j,k,i,l	30.100
				TCacheTurns	k,j,i,l	3.250
				Compiler	compiler selection	30.100
mgrid	psinv	150,150,150	150,150,150	Loopcost	k,j,i	302.340
				TCacheTurns	j,k,i	4.150
				Compiler	compiler selection	302.340
swim	calc3	1000,1000	1335,1335	Loopcost	j,i	29.600
				TCacheTurns	j,i	29.600
				Compiler	compiler selection	29.600

The results with the native compiler are also worse, in many cases, than our algorithm.

6. Conclusions

In this paper we presented an algorithm to determine how to arrange the loops in a nest so as the number of cache misses is minimized due to array self-interferences. It is based on a novel model of the cache, where parameters were defined to give information about the cache memory occupation

when affine array references to main memory are issued from a nested loop. Our algorithm (*TCacheTurns*) was compared with the well-known *loopcost* algorithm, obtaining similar or better results. We have shown how a simple model of a cache allows to determine a very effective arrangement of loops in a nest that optimizes the occupation of the cache. As a future work we will study how to define a general formal framework to apply a set of important compiler optimization techniques, like padding, tiling, loop reversal, and so on.

References

[1] J. R. Allen and K. Kennedy: *Optimizing Compilers for Modern Architectures: A Dependece-based Approach*. **Morgan Kaufmann Publishers**. October 2001.

[2] D.F. Bacon, S.L. Graham and O.J. Sharp: *Compiler Transformations for High-Performance Computing*. **ACM Computing Surveys**, Vol. 26, No. 4, pp. 345-420. December 1994.

[3] D.H. Bailey: *Unfavorable Strides in Cache Memory Systems*. **Technical Report RNR-92-015**, NASA Ames Research Center, CA. 1992.

[4] S. Carr, K.S. Mckinley and C. Tseng: *Compiler Optimizations for Improving Data Locality*. **In Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems**, San Jose, CA. October 1994.

[5] Ph. Clauss and B. Meister: *Automatic Memory Layout Transformation to Optimize Spatial Locality in Parameterized Loop Nests*. **ACM SIGARCH Computer Architecture News**, Vol. 28, No. 1. March 2000.

[6] Ph. Clauss, V. Loechner and B. Meister: *Minimizing strides in Loops with Affine Array References*. **In Proceedings of the Compilers for Parallel Computers**, Edinburgh, Scotland. June 2001.

[7] P. D'Alberto, A. Nicolau, A. Veidenbaum and R. Gupta: *Static Analysis of Parameterized Loop Nests for Energy Efficient Use of Data Caches*. **Compilers and Operating Systems for Low Power**, pp. 193-207, Norwell, MA, USA, Kluwer Academic Pub. 2003.

[8] S. Ghosh, M. Martonosi and S. Malik: *Cache Miss Equation: An Analytical Representation of Cache Misses*. **In Proceedings of the 11th International Conference on Supercomputing**, Vienna, Austria. 1997.

[9] S. Ghosh, M. Martonosi and S. Malik: *Cache Miss Equations: A Compiler Framework for Analyzing and Tunning Memory Behaviour*. **ACM Transactions on Programming Language and Systems**. July 1999.

[10] M. Kandemir, P. Banerjee, A. Choudhary, J. Ramanujam and E. Ayguade: *An Integer Linear Programming Approach for Optimizing Cache Locality*. **In Proceedings of the 13th International Conference on Supercomputing**, Rhodes, Greece. June 1999.

[11] M. Kandemir and J. Ramanujam: *Data Relaton Vectors: A New Abstraction for Data Optimizations*. **IEEE Transactions on Computers**, Vol. 50, No. 8. August 2001.

[12] V. Loechner, B. Meister and Ph. Clauss: *Precise Data Locality Optimization of Nested Loops*. **The journal of Supercomputing**, Vol. 21, No. 1, pp. 37-76, Kluwer Academic Pub. 2002.

[13] M. O'Boyle and P. Knijnenburg: *Integrating Loop and Data Transformations for Global Optimizations*. **IEEE Int'l. Conf. on Parallel Architectures and Compilation Techniques**, Paris, France. October 1998.

[14] Y. Paek, J. Hoeflinger and D. Padua: *Simplification of Array Access Patterns for Compiler Optimizations*. **In Proceedings of the SIGPLAN Conference of Programming Language Design and Implementation**. June 1994.

[15] A.D.B. Weikle, S.A. McKee, K. Skadrom and W.A. Wulf: *Cache As Filters: A Framework for the Analysis of Caching System*. **In Proceedings of th Grace Murray Hopper Conference 2000**. September 2000.

Performance Evaluation of Barrier Techniques for Distributed Tracing Garbage Collectors

Jose M. Velasco[†], David Atienza[†⋆], Katzalin Olcoz[†], Francky Catthoor[*]
[†]DACYA/UCM, Avda. Complutense s/n, 28040 Madrid, Spain.
Email: mvelascc@fis.ucm.es, {datienza, katzalin}@dacya.ucm.es
[⋆]IMEC vzw, Kapeldreef 75, 3001 Heverlee, Belgium.
Email:{Francky.Catthoor, atienza}@imec.be

Abstract. Currently, software engineering is becoming even more complex due to distributed computing. In this new context, portability is one of the key issues and hence a cluster-aware Java Virtual Machine (JVM) that can transparently execute Java applications in a distributed fashion on nodes of a cluster, while providing the programmer with the single system image of a classical JVM, is really desirable. This way multi-threaded server applications can take advantage of cluster resources without increasing their programming complexity.

However, such kind of JVM is not easy to design. Moreover, one of the most challenging tasks in its design is the development of an efficient, scalable and automatic dynamic memory manager. Inside this manager, one important module is the automatic recycling mechanism or garbage collector. This collector is a module with very intensive processing demands that must concurrently run with user's application. It can consume a very significant portion of the total execution time spent inside JVM in uniprocessor systems, and its overhead increases in distributed garbage collection because of the update of changing references in different nodes. Hence, the garbage collector is a very critical part in distributed designs of JVMs, both for performance and energy.

In this paper our contribution to automatic distributed garbage collection is two-fold. First, we have analyzed the barrier mechanism design space for the study of tracing-based distributed garbage collectors. Second, we have evaluated the impact of the most significative barrier strategies as main bottlenecks in global performance. Our preliminary results show that the choice of the specific technique used in barrier mechanisms produces significant differences both in performance and inter-nodes messaging overhead.

1. Keywords

Software engineering, clusters, runtime support, distributed garbage collection.

2. Introduction

A cluster-aware Java Virtual Machine (JVM) can transparently execute java applications in a distributed fashion on the nodes of a cluster while providing the programmer with the single system image of a classical JVM. This way multi-threaded server applications can take advantage of cluster resources without increasing the programming complexity.

When a JVM is ported into a distributed environment, one of the most challenging tasks is the development of an efficient, scalable and fault-tolerance automatic dynamic memory manager. The automatic recycling of the memory blocks that are no longer used is one of the most attractive characteristics of Java for the programmer and the software engineer since engineers do not need to worry

[*]This work is partially supported by the Spanish Government Research Grant TIC2002/0750.

about designing a convenient dynamic memory management, i.e. missusing the available memory because incorrect intervals for memory allocation and deallocation are provided. This automatic process, very well-known as Garbage Collection (GC), makes much easier the development of complex parallel applications that include different modules and algorithms with different dynamic memory behaviors that need to be taken care of from the software engineering point of view. However, since the garbage collector is an additional module with intensive processing demands that runs concurrently with the application itself, it always attains for a critical portion of the total execution time spent inside the virtual machine in uniprocessor systems. As Plainfosse and Shapiro[7] point out, distributed GC is even harder because of the difficult job to keep updated the changing references between the address spaces of the different nodes.

In this paper we consider a distributed memory heap partitioned into disjoint spaces. Spaces communicate with each other by message passing and the allocation can be local or remote. Spaces records the entering references and use them as living roots for tracing purposes. In addition, in our approach the global marking phase is concurrently executed with the application.

Intuitively, the weakness of tracing algorithm relies in the cost of barriers and lack of scalability due to the amount of messages going through the processing nodes. In fact, our preliminary results show that the choice of the technique used in write-barriers produces significant differences both in performance and inter-nodes messaging overhead.

This rest of the paper is organized as follows. We first describe related work on both distributed garbage collection and uniprocessor barrier techniques. Link to this, we then overview the main topics in distributed tracing garbage collection. Next, we present our proposed barrier techniques design space and the configurations we have explored in this paper. Later, we describe the experimental setup used in our experiments and the results obtained. Finally, we present an overview of our main conclusions and outline possible future research lines.

3. Related Work

In this section, we discuss related work on both distributed garbage collection and uniprocessor barrier techniques. To our knowledge, no one has developed a experimental comparison among barrier techniques in a distributed context.

Plainfosse and Shapiro[7] published a complete survey of distributed garbage collection techniques. For good tutorial overview, Lins offers a chapter within the Jones's classical book about garbage collection [4].

In section 6.2 we make a presentation of dJVM from Zigman et Al. A distributed Java virtual machine on a cluster which presents a single system image to the programmer. Another similar aproach is the Java VM on a cluster from Aridor et Al[9]. This distributed virtual machine is built on top of a cluster enabled infrastructure.They have developed a new object model and thread implementation that are hide to the programmer.

In [6], Pirinen presents a methodical analysis of barrier techniques from a formal point of view and for incremental tracing. Our present work is an extension of his. In this paper, we have implemented the different options in the much stricter environment of distributed collection to make an empirical study and to look for optimal solutions.

Blackburn et Al[1], analyze the effect of inlining on write-barriers. They use *Remembered sets* with slots and objects and they measure the impact of inlining and partial inlining on execution time and compilation overload. They use Jikes RVM with the optimizing compiler.

4. Distributed Tracing Garbage Collection

In prior work, the cluster JVM from Aridor et Al([9]) and the distributed JVM (i.e. dJVM) from Zigman et Al([10]) use a conservative approach that does not reclaim objects with direct or indirect global references. In our research, we are developing a new framework for the analysis and optimization of tracing-based distributed garbage collectors by using the dJVM as background approach. Our new framework does not include any reference counting collection mechanisms [5], which are very popular in monoprocessor GC solutions, because of two reasons. First, the reference counting collector is not complete and needs a periodical tracing phase to reclaim cycles, which will create an unaffordable overhead in execution time since it needs to block all the processing nodes. Second, the update of short-lived references produces continuous messages to go back and forth between the different nodes of the distributed GC, which makes this algorithm not scalable within a cluster.

On the contrary, as we propose, tracing collectors seem to be the more convenient for creating such distributed garbage collectors. Conceptually, all consist in two different phases([5]), which are the following:

- First, the marking phase allows the garbage collector to identify living objects. This phase is global and implies scanning the whole distributed heap.
- Second, the reclaiming phase takes care of recycling the unmarked objects (i.e. garbage). The reclaiming phase is local to each node and can be implemented as a non-moving collector or as a moving collector.

During the mark phase, the collector traverses the graph of references between objects through memory recursively. As this phase makes progress, the collector segregates the objects into three sets:

- Black objects which have been marked as alive and will not be visited again during the tracing. We say that they are scanned.
- Grey objects have been noted reachable but must still be processed in order to follow the graph to their offspring. When a white object becomes grey, we call this *shading*.
- White objects have not been visited and are candidates to be recycled.

If this phase is interleaved with the program execution, the garbage collector needs to keep track of the changes in the reference graph produced by the application. To guarantee that no living object will be reclaimed, the collector must be aware of every new reference created from a black object to a white object. To this end, the JVM executes barriers to remember the locations where new pointers have been created or modificated. Therefore, the main problem associated with distributed tracing comes from the difficult synchronization between the distributed global mark phase with several local and independent recycling phases.

In the uniprocessor context, both for incremental and generational garbage collectors, it has been proven that write-barriers are the main bottleneck that degrades performance, and we believe that in distributed environments, the optimization of write-barriers is even more critical. In summary, barriers are used to intercept new references among objects and give the virtual machine a chance to be aware of this situation before it is concluded. *Read-barriers* intercept loads and *write-barriers* intercept stores. There are several techniques for maintaining the tricolour invariant. Each of them uses read-barriers or write-barriers or both.

In his paper, Pirinen define two invariants relative to the tricolour marking phase:

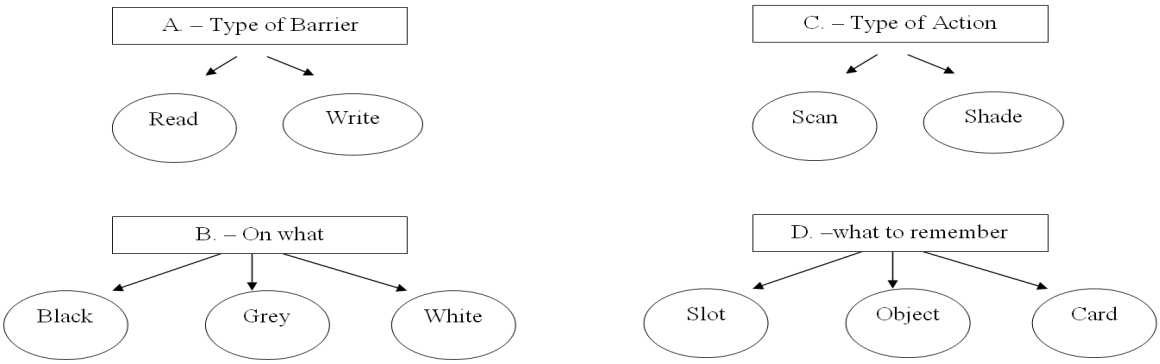


Figure 1. Design Space of Orthogonal Decisions for Barrier Techniques

- Strong invariant: There are no pointers from a black object to a white object.
- Weak invariant: All white objects pointed to by a black object are reachable from some grey object through a chain of white objects.

5. Design Space of Orthogonal Decisions for Barrier Techniques

As we have just explained, an extensive amount of possible barrier techniques (and implementations for them) exists. Therefore, all these options have to be enumerated to cover exhaustively the barriers design space. In our method, we have classified all the relevant decisions that can compose the design space of barrier mechanisms in different orthogonal decision trees (see Figure 1). Orthogonal means here that any decision in any tree can be combined with any decision in another tree, and the result should be a potentially valid combination, which does not necessarily mean that it meets all timing and cost constraints for a specific system. In addition, all possible solutions in the design space should be spanned by a combination of leaves in the orthogonal trees, just like any point in a geometrical space can be represented in a set of orthogonal axes. Moreover, the decisions in the different orthogonal trees can be ordered in such a way that traversing the trees can be done without decision iterations, as long as the appropriate constraints are propagated from one decision level to all subsequent levels. Basically, when one decision has been taken in every tree, one custom barrier techniques is defined (in our notation, atomic barrier mechanism) for a specific memory manager behaviour pattern. As a result, the four trees considered in our design space are the following ones:

- A. Type of Barrier: barriers are used to intercept the accesses among objects and give the virtual machine a chance to be aware of this situation before it is concluded. *Read-barriers* intercept loads and *write-barriers* intercept stores. There are several strategies for maintaining the tricolour invariant. Each of them uses read-barriers or write-barriers or a combination of both.
- B. On what type of objects the barrier is set. In this case the type of object is defined in terms of the tricolour marking scheme.
- C. When a barrier is hit what kind of action we must take: turn the object under the barrier black (*scan*) or turn the referent grey (*shade*). In this paper we have experimented taking only the *shade* option.

- D. As a result of the barrier, how we remember the relationship graph change:
  - Slot. We remember the object field that contains the pointer. The set of addresses of these fields are usually knom as *Remembered Set*. In this paper we have only explored this option since it is the one that seems to achieve more performance in the final distributed GCs.
  - Object. We remember the source object itself. The collector need to scan the whole object in order to find references.that the choice of the specific technique used in write-barriers produces significant differences in both performance and inter-nodes messaging overhead.
  - Card marking. This mechanism uses a table to remember fixed size of regions of memory (*cards*) as pointer sources.During collection the virtual machine needs to scan these memory regions looking for pointers.

Although the decision categories and trees presented in Figure 1 are orthogonal, certain leaves in some trees strongly affect the coherent decisions in other trees. Thus, they include interdependencies to take into account when a barrier techinque is designed. This fact, jointly with the mentioned choices for this paper that we have taken in C and D trees, have produced four barrier configurations:

- C1. A read-barrier on grey objects.
- C2. A write-barrier on black objects.
- C3. A write-barrier on both grey and white objects.
- C4. A write-barrier on grey objects and a read-barrier on white objects.

The two first configurations maintain the *Strong* invariant relative to tricolour marking scheme, while the two next ones preserve the *Weak* invariant.

6. Experimental Setup and Results

In this section we first describe the whole simulation environment used to obtain detailed memory access profiling of the JVM (for both the application and the collector phase). It is based on cycle-accurate simulations of the original Java code of the applications under study. Then we summarize the representative set of GCs used in our experiments. Finally we introduce the sets of applications selected as case studies and indicate the main results obtained with our experiments.

6.1. Basic Jikes RVM

Jikes RVM is a high performance Java virtual machine designed for research. It is written in Java and the components of the virtual machine are Java objects [3], which are designed as a modular system to enable the possibility of modifying extensively the source code to implement different GC strategies, optimizing techniques, etc. We have used version 2.3.0 along with the recently developed memory manager JMTk (Java Memory management Toolkit) [2]. Jikes RVM offers three compiler choices:

- Baseline, all methods are compiled by a quick non-optimizing compiler.
- Optimizing, all methods are compiled by an aggresive optimizing compiler.

- Adaptive, which initially compiles methods with the quick compiler, and then and after selecting hot zones, recompiles methods using the optimizng compiler. The hot method identification is based on application sampling and therefore it tends to produce slightly different choices for each compilation.

At runtime Jikes RVM compiles not only the application classes, but also the virtual machine classes. Since several classes are needed for bootstrapping the VM, Jikes can be configured with two levels of VM classes precompilation at build-time:

- A minimal configuration that only precompiles those classes essentials for booting the VM.
- A fast running configuration that precompiles as much as possible, including key libraries and the optimizing compiler.In this paper we have used this configuration.

Jikes code is scattered with a lot of assertion checking code that we have disabled for our experiments.

6.2. Distributed Jikes RVM

We have employed as our baseline framework to modify the Distributed Jikes RVM (dJVM)[10] developed at the Australian National University. It provides a single system image to Java applications and so it is transparent to the programmer. The dJVM employs a master-slave architecture, where one node is the master and the rest are slaves. The boot process starts at the master node. This node is also responsible for the setting up of communication channels among the differents slaves. The class loader runs in the master. In dJVM are objects available remotely and object which have only a node local instance. This is achieved by using a global and a local addressing schemes for objects. The global data is also stored in the master with a copy of its global identifier in each slave node. Each object has an associated universal identifier (UID) that uniquely identifies the object in the whole cluster. Each node has a range of UIDS of its own. Instances of primitives types, array types and most class types are always allocated locally. The exceptions are class types which implement the *Runnable* interface.

As the goal of this paper was the measurement of different barrier techniques, we have changed dJVM in order to force a bigger amount of remote object allocations. This way the performance is degraded but it gives us a better opportunity for analyzing the barriers influence.

6.3. Case Studies

We have applied the proposed experimental setup to dJVM running the most representive benchmarks in the suite SPECjvm98 [8]. These benchmarks could be launched as dynamic services and extensively use dynamic data allocation. The used set of applications is the following:

_201_compress: it compresses and then decompresses a large file.

_202_Jess: it is the Java version of an expert shell system using NASA CLIPS. It is compound fundamentally of structures of sentences if-then.

_205_Raytrace: raytraces a scene into a memory buffer. It allocates a lot of small data with different lifetimes. [4] _209_db: This benchmark reads a 1 MB size file and then it performs multiple database functions on memory.

_213_javac: it is the java compiler. It has the highest program complexity and its data is a mixture of short and quasi-inmortal objects. problem _222_mpegaudio: it is an MPEG audio decoder.

_227_Jack: it is a parser based on the Purdue Compiler Construction Tool Set (PCCTS). A parser determines the syntactic structure of a chain of symbols received from the exit of the lexical analyzer.

Table 1
Summary of results, normalized against C1 (read-barrier on grey objects)

	Execution Time				Messaging Overhead				Remembered Set Size			
	C1%	C2%	C3%	C4%	C1%	C2%	C3%	C4%	C1%	C2%	C3%	C4%
compress	100	94.4	85	95.5	100	92.2	77	83	100	90	50	76
Jess	100	92.4	73	79.9	100	94.4	66	71.5	100	91.6	84	77.5
Raytrace	100	91.6	84	77.5	100	96.3	55	75.5	100	92.4	73	79.9
db	100	98.4	87	75.5	100	94.4	73	72	100	94.4	88	72.5
javac	100	95.2	69.5	78.3	100	92.4	58	75.5	100	94.6	75	83
mpegaudio	100	94.4	88	72.5	100	93.4	55	75.5	100	94.4	55	75
Jack	100	94.6	75	83	100	94.4	68	79	100	90.8	64	75.8
mtrt	100	90.8	64	75.5	100	94.4	67	77	100	94.4	55	75.5

_228_mtrt: it is the multi-threaded version of _205_raytrace. It works in a graphical scene of a dinosaur. It has two threads, which make the render of the scene removed from a file of 340 KB.

The suite SPECjvm98 offers three input sets(referred as s1, s10, s100), with different data sizes. In this study we have used the biggest input data size, represented as s100, as it produces a bigger amount of cross-references among objects.

6.4. Experimental Results

In our experiments we have utilized as hardware platform an eight nodes cluster with Fast-Ethernet communication hardware. The networking protocol is TCP/IP. Each node is a Pentium IV, 866MHz with 1024Mb and Linux Red Hat 7.3. In Table 1 we summarize our experimental results. As we can see, the dichotomy between Strong and Weak invariant is resolved in favour of the latter. As a result, the C3 and C4 configurations obtain always better results both in execution time and in inter-node messaging overhead. Furthermore, for those configurations based on full or partial read-barriers, our results indicate a significant decrease on performance due to the unnecessary messages sent between the nodes, in comparison to write-barriers only schemes. Therefore, the configuration C3, with a write-barrier on both grey and white objects achives the better results. Following this previous reasoning of limiting the amount of messages per processed data, the benefits of C3 are shown even more clearly on those benchmark with more allocated data as _205_Raytrace or _228_mtrt.

7. Conclusions and Future Work

The barrier technique is a key factor relative to performance in uniprocessor garbage collectors. Related to this well-known problem, in a distributed context the increase of inter-node messaging problem is added. In this paper we have first presented the design pace of orthogonal decisions for barrier techniques. Second, we have evaluated four representative barrier mechanism configurations. Two of them maintain the *Strong* tricolour marking scheme invariant, and two that preserve only the *Weak* invariant to achieve faster execution. Our preliminary results show that the choice of the specific technique used in barrier mechanism produces significant differences in both performance and inter-nodes messaging overhead.

As future work we intend to extend our experimental results to different configurations, some of them maybe not even considered before in distributed garbage collection due to their limited design space, in order to cover the C and D trees of our complete barrier techniques' design space.

Acknowledgements

This work is partially supported by the Spanish Government Research Grant TIC2002/0750.

References

[1] Stephen M. Blackburn and Kathryn S. McKinley. In or out? putting write barriers in their place. In *Proceedings of SIGPLAN 2002 International Symposium on Memory Managment, ISMM'02, Berlin, June, 2002*, ACM SIGPLAN Notices. ACM Press, June 2002.

[2] IBM. The jikes' research virtual machine user's guide 2.2.0., 2003. `http://oss.software.ibm.com/developerworks/oss/jikesrvm/`.

[3] The source for java technology, 2003. `http://java.sun.com`.

[4] Richard Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, 4th edition, July 2000.

[5] Richard Jones and Rafael D. Lins. *Garbage Collection : Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, 1996.

[6] Pikka Pirinen. Barrier techniques for incremental tracing. In *Proceedings of International symposium on Memory Management*, Vancouver, Canada, September 1998.

[7] David Plainfoss and Marc Shapiro. A survey of distributed garbage collection techniques. In *Proceedings of International Workshop on Memory Management*, September 1995.

[8] SPEC. Specjvm98 documentation, March 1999. `http://www.specbench.org/osg/jvm98/`.

[9] Michael Factor Yariv Aridor and Avi Teperman. A distributed implementation of a virtual machine for java. *Concurrency and Computation: practice and experience*, 13:221–244, 2001.

[10] John Zigman and Ramesh Sankaranarayanara. djvm - a distributed jvm on a cluster. Technical report, 2002.

A new Strategy for Shape Analysis based on Coexistent Link Sets *

A. Tineo[a], F. Corbera[a], A. Navarro[a], R. Asenjo[a], and E.L. Zapata[a]

[a]Dpt. of Computer Architecture, University of Málaga,
Complejo Tecnológico, Campus de Teatinos, E-29071. Málaga, Spain.
{tineo,corbera,angeles,asenjo,ezapata}@ac.uma.es

The analysis of dynamic heap-based data structures is difficult due to the alias problem. Shape analysis tries to gather information conservatively about these structures at compile time. In the context of parallelizing compilers, information about how memory locations are arranged in the heap at runtime is essential for data dependence analysis. With proper shape information we can reveal parallelism for heap-based structures, which are typically ignored by compilers. Existing shape analysis techniques face a dilemma: either they are too costly to be useful for real compilers or they are too imprecise to be useful for real programs. In this work, we present a new strategy for shape analysis based on a compact representation for the shape of data structures. This is done by using *Coexistent Link Sets* for nodes in a graph. The technique is simple to implement and very precise at the core level. Further precision-vs-cost balance can be tuned with the use of extensible properties.

1. Introduction

Static knowledge of memory references in a program is a must for compilers, if they are to provide optimizations related to parallelism in an automated basis. Such knowledge is not easy to gather due to the existence of aliases. Arrays, pointers and pointer-based dynamic data structures introduce aliases in programs. Parallelizing compilers have obtained a reasonable degree of success when dealing with array aliases and stack-directed pointers. However, heap-directed pointers and the structures they dereference are a whole different ground that still needs significant work.

The problem of characterizing dynamically allocated memory locations in a program can be approached in several ways. We believe that, in order to provide accurate information for real-life programs, some sort of abstraction in the form of a bounded graph must be performed. The kind of analysis that represents the heap as a storage shape graph is known as *shape analysis*. Its main goal is to capture the *shape* of memory configurations that are accessible through heap-directed pointers in a program.

Information about the shape of dynamic data structures is useful for parallelizing compiler transformations over the input program. Maybe the most obvious application is data dependence detection, needed for instruction scheduling, data-cache optimization, loop transformation and automatic vectorization and parallelization. Another interesting application comes from the use of the shape information for debugging analysis of the program. The shape abstraction can provide information about incorrect pointer usage that can lead to mistakes difficult to track.

We present in this work a new strategy for shape analysis based on what we call *Coexistent Link Sets* (CLSs). CLSs codify possibilities of connectivity between memory locations in a

*This work was supported in part by the Ministry of Education of Spain under contract TIC2003-06623.

neat and compact way. This is done by using graphs that represent the possible states of the memory configuration at a program point. Information is kept as a combination of possible reaching and leaving links over the memory locations. CLSs provide a rich description of the data structure with little storage requirements.

The goal of this paper is to present a new technique to achieve a shape abstraction of dynamic data structures. We think that it will provide more precision than existing techniques while at the same time keeping the storage and computation cost at a reasonable level. CLSs are the key instruments for the development of this technique. The remainder of this paper is organized as follows: Section 2 introduces the basics for our shape analysis technique; Section 3 describes CLSs in greater detail; Section 4 explains our criteria for summarization in graphs; Section 5 describes how the shape analyzer works by example; Section 6 comments some related work; and finally Section 7 concludes with the main contributions and ideas for future work.

2. Shape analysis basics

Our approach to shape analysis is based on graphs. A program dealing with dynamic data structures performs allocation of *memory locations* and establishes links between them. We represent memory locations in the program as nodes in graphs. Nodes can be referenced by pointer variables through *pointer links* (PLs). Additionally, *selector links* (SLs) are used to link nodes with other nodes.

The size of dynamic data structures is undecidable at compile time. Therefore, we must provide some mechanism to sum up the possible memory configurations in a finite, *bounded shape graph*. To achieve this, we can summon nodes that can actually represent several memory locations that are similar. We call this kind of nodes, *summary nodes* and the process of merging similar nodes, *summarization*. Very often, however, some of those locations are accessed later in the program and become so-called *singular* locations, which are somewhat different to other locations in the structure. It would be desirable to provide a mechanism to *invert* the summarization process, i.e., we would like to be able to *focus* over a singular node extracted out of a summary node. This can be achieved with the *materialization operation*. Depending on the case, we will be able to recover the information as we had it or instead, a conservative and less precise node will be materialized.

In our approach, there is a single graph associated to every statement in the program, which represents the possible memory configuration states at that point in the program. The shape analyzer works as an iterative data-flow analysis, by symbolically executing the statements in the source program, a process called *abstract interpretation*. For example, pointer statements receive an input graph (SG_{in}) and modify it to produce an output graph (SG_{out}). The rules for such transformation are determined by the statement *abstract semantics*.

Our technique only cares about statements that involve operations through pointers (pointer statements) and control flow decisions (loop and branch statements). Fig. 1 sketches out how the analysis operates in the presence of these kind of statements in a general, descriptive way.

In order to simplify the analysis, pointer statements are normalized to *simple* pointer statements, i.e., those that contain only simple access paths, or 1-level indirection. The simple pointer statements are (in C syntax):

```
ptr=NULL;         ptr=malloc(...);  ptr1=ptr2;
ptr1->sel=ptr2;   ptr1=ptr2->sel;   ptr->sel=NULL;
```

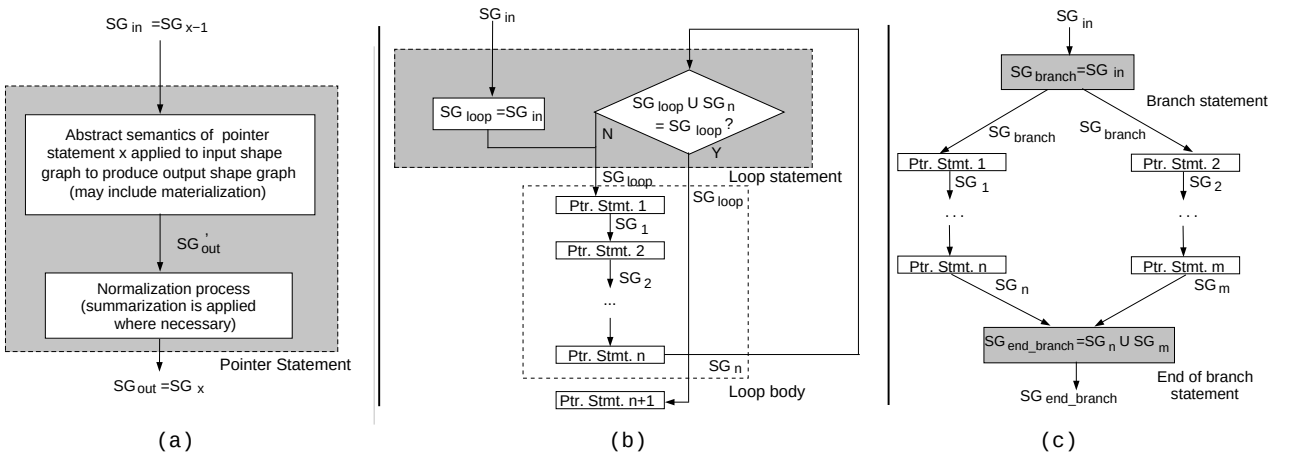


Figure 1. Analysis operation in presence of (a) pointer statements, (b) loop statements and (c) branch statements.

At loop bodies the analysis must iterate over the statements of the loop until the graphs for each statement change no more, i.e., until the analysis achieves a *fixed-point*. This way we ensure that the graph for each statement holds all possible memory configurations at that point of the program at runtime. When a fixed-point is reached for all statements in the program, the analysis terminates. Termination of the algorithm is assured by the summarization process that automatically occurs whenever nodes are similar. The graphs cannot grow out of control and eventually the graphs will be bounded and stationary.

At join points in the CFG, such as loops and branching statements, *incompatible* memory configurations occur. An operation to join graphs, (*graph union*) while conservatively keeping all possibilities is needed. For instance, fig. 2(b) shows the two possible memory configurations (`MC1` and `MC2`) at the end of the program in fig. 2(a). In the next section we will see how these two MCs can be represented in one graph.

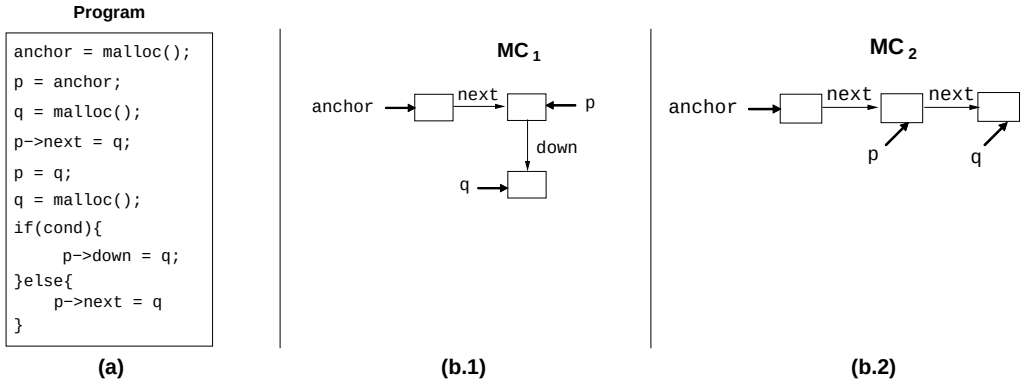


Figure 2. There are two memory configurations for program (a), namely (b.1) and (b.2).

3. Coexistent Link Set (CLS)

Our main contribution to shape analysis in this paper is the introduction of *Coexistent Link Sets* (CLSs). To better help us describe what a CLS is, let us consider an example. Fig. 3 shows the graph that would represent the memory configurations shown in fig. 2.

Let us assume at this point that we need three nodes (`n1`, `n2` and `n3`) for the graph that captures `MC1` and `MC2` from fig. 2. In Section 4 we describe in detail why this is so. We could

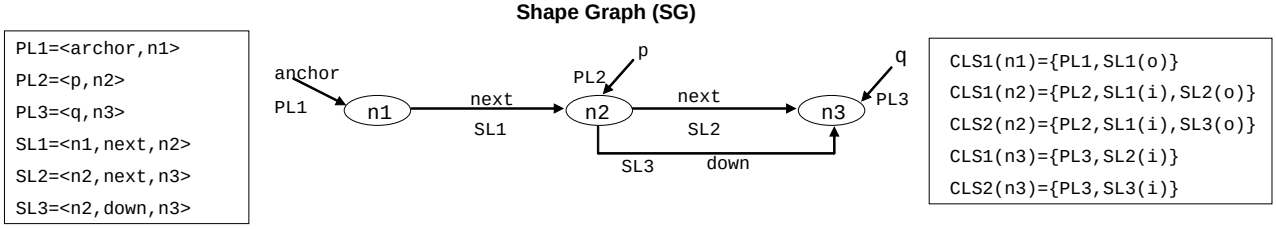


Figure 3. Shape graph for example of fig. 2. CLSs describe the shape abstraction.

expose a hierarchical view of the graph: the base elements are nodes, pointers and selectors; pointers and nodes can be combined to create pointer links (PLs); nodes and selectors can be combined to create selector links (SLs); finally pointer links and selector links can be combined to create coexistent link sets (CLSs).

CLSs allow us to express possibilities of connectivity between nodes, i.e., they describe the links that may reach and leave a node in the memory configuration abstraction. In the example of fig. 3, CLS1(`n1`) is telling us that `n1` supports `PL1=<anchor,n1>` and `SL1=<n1,next,n2>`. No other combination of links is possible for this node: `n1` can only be reached through the pointer `anchor` and connects undoubtedly to `n2` through its `next` selector. For `n2`, there are two possibilities. In any of them, `PL2=<p,n2>` reaches the node (i.e., `p` points to `n2` in any case) and the node is reached from `n1` through selector `next` (SL1). However, `n2` can follow to `n3` through the `next` selector or through the `down` selector, yielding two possibilities of connectivity for this node: CLS1(`n2`) and CLS2(`n2`). CLSs for `n3` work in a similar way: `n3` always supports `PL3=<q,n3>` and can be reached from `n2` by following either SL2 or SL3.

SLs are stand-alone entities in the graph. They represent links between nodes through selectors. However, when used in the context of a CLS, they are complemented with information about the origin and destination nodes. Thus, SLs within CLSs can be: 1) *incoming* `(i)`, when the SL is a *reaching* link for the current node; 2) *outgoing* `(o)`, when the SL is a *leaving* link for the current node; 3) and *cyclic* `(c)`, when the SL is at the same time a reaching and leaving link for the current node. In addition, a SL can be regarded as *shared*, if it can be used more than once to reach the same destination node for a given CLS. In our notation, a SL is not shared unless labelled with `(s)`.

4. Summarization criteria

In the previous example we asked the reader to assume that three nodes were needed for the shape abstraction representation. We now explain the summarization criteria for our shape analysis technique. This criteria determines to a great extent the behavior and precision of the analysis. First, we define a basic, fixed mechanism about what nodes should be kept separate and what nodes *could* be merged; second, we add configurable *properties* to fine-tune summarization decisions.

The basic criterion for summarizing nodes dictates that nodes that are pointed by the same (possibly empty) set of pointers are merged together, i.e., if P(`n1`)=P(`n2`) then `n2` merges into `n1`. Nodes that are pointed by pointers always remain singular nodes, i.e., they represent only one memory location for a given memory configuration. On the contrary, nodes that are not pointed by pointers will be merged (according to this basic criterion) into a unique summary node, that may be representing more than one actual memory location. This basic idea allows us to achieve precision on the *entry points* to the data structures, where it is more likely to be needed. This is the criterion applied in fig. 3.

However, merging all locations that are not directly accessed by pointers in a single node

can be very imprecise as soon as the data structures are a little complex, and this is certainly the case for real programs. *Node properties* can be used to prevent *too much* summarization in such cases. A compatibility function must be defined for every property. If two nodes are *compatible* with respect to all available properties and conform to the basic criterion (share the same pointers), then they will be merged. Otherwise they will be kept separate. This way there could be several summary nodes, while in the case of using only the basic criterion, there would be just one summary node for the whole graph.

Properties are configurable, in the sense that they can be *turned on* and *off* at will, depending on the requirements of the analysis. It is clear to see that keeping a lot of properties will yield bigger graphs, with more nodes, and this will have an impact in the analysis cost. On the other hand, properties are absolutely needed to carry precision for analysis of complex structures.

In particular, the *touch* property is a key instrument for data dependence analysis. While CLSs on their own capture *spatial* or *topological* information, the *touch* property is used to capture *temporal* information. During a typical structure traversal, locations are accessed for reading or writing. From a data dependence analysis point of view, it is of key importance to be able to discriminate, in the course of the traversal, between accessed (we say *touched*) locations in a structure and locations that have not been yet accessed (*untouched*). The touch property is annotated in nodes to avoid merging visited nodes with unvisited ones. Other properties can be effortlessly added, to add information about data type, structure connection, allocation site, etc., similarly to [1].

5. Shape analysis operations

In this section we illustrate how the shape analyzer works by example. First, we analyze a program that creates a single-linked list. Due to space limitations we will just focus on certain key aspects that are needed to understand the process. In fig. 4(a) we present the program used to create the list and the first steps of the algorithm.

Input graph is empty. Abstract interpretation of statement `S1` produces the creation of node `n1` and the corresponding PLs, SLs and CLSs. Notice that the analysis can track uninitialized selectors. This is useful for code correctness checking. Upon entering a loop for the first time (`S3` in this example), input graph is preserved. Then, in `S4`, a new element is created and PLs, SLs and CLSs are updated accordingly. Changes with respect to previous graph appear in bold. Graphs for statements inside a loop are labelled with a superscript showing the number of the symbolic iteration.

Let us now consider an interesting situation that appears later in the analysis. Upon interpreting `S5` for the third time (third iteration), we obtain the shape graph shown in fig. 4(b.1). At this point the list can be two (`n6` and `n3`), three (`n1`, `n4` and `n3`) or four elements (`n1`, `n2`, `n4` and `n3`) long. Two nodes exist for the first element of the list to account for incompatible configurations.

After applying the abstract semantics of `S6` in fig. 4(b.2), we find that nodes `n1-n6` and `n2-n4` now meet the summarization basic criterion, i.e., they are pointed by the same (possibly empty) set of pointers (P(`n1`)=P(`n6`)={list}, P(`n2`)=P(`n4`)={$\emptyset$}), so they must be merged. For this example we do not consider properties for summarization. The process is completed in fig. 4(b.3), where the shape graph has been *normalized* to conform to the selected rules of summarization. The normalization process involves five steps: 1) find nodes that meet the current summarization criteria (`n1-n6` and `n2-n4`); 2) substitute all references of the nodes to

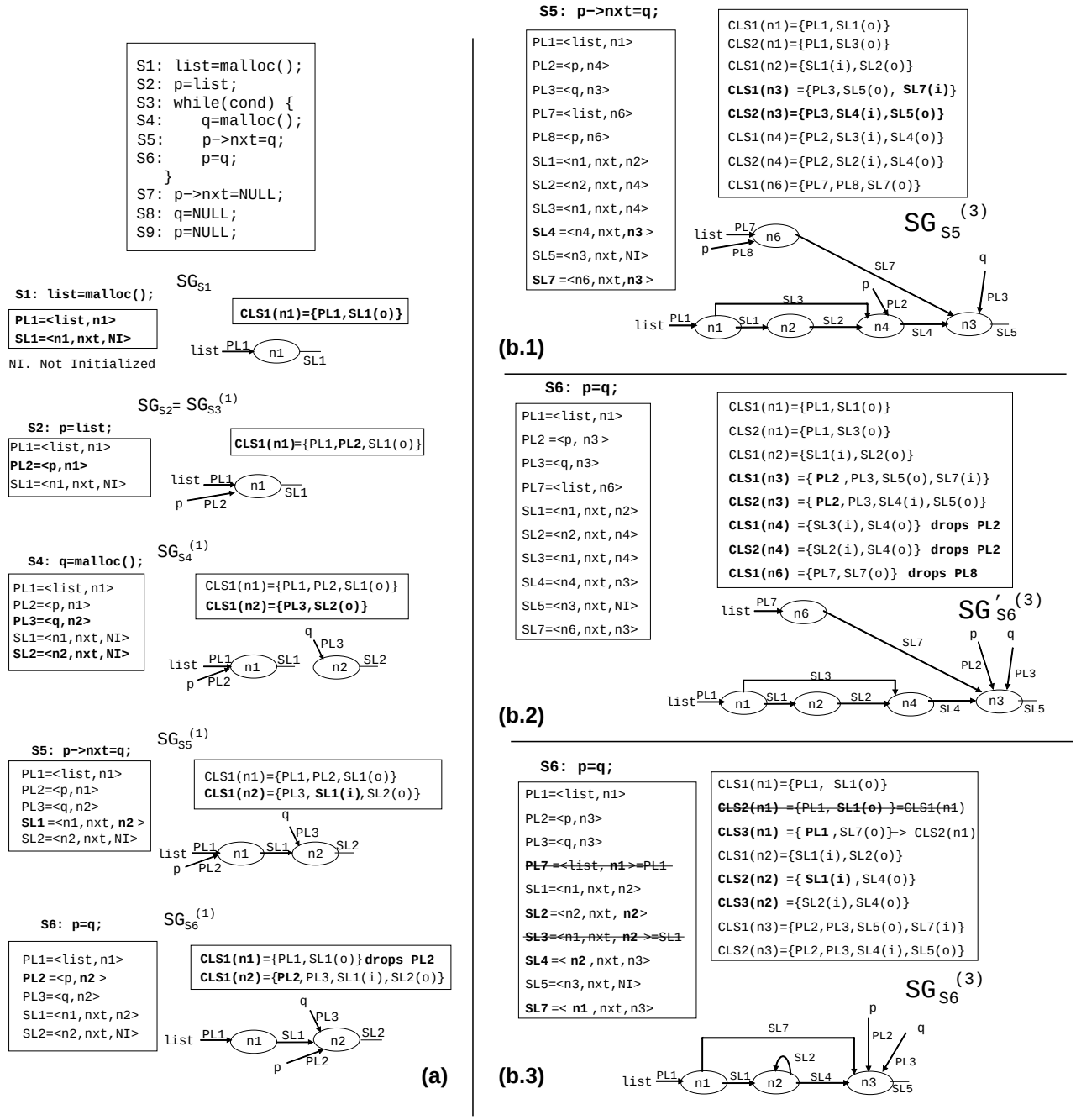


Figure 4. (a) Start of analysis of single-linked list; (b) Node summarization in 3 steps.

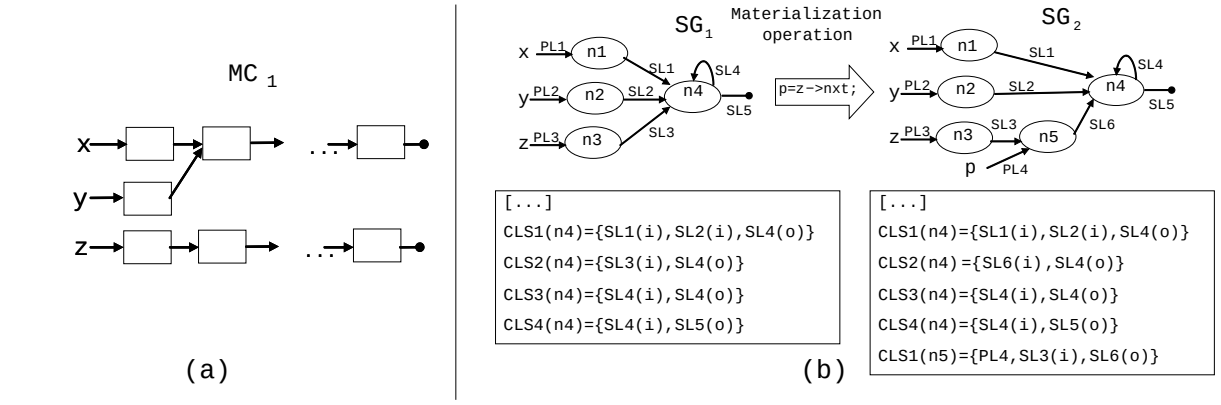


Figure 5. (a) Memory configuration of shared and non-shared lists; (b) Materialization ex.

be merged by one of them in the lists of PLs and SLs (substitute `n6` for `n1` in PL7 and SL7 and substitute `n4` for `n2` in SL2, SL3 and SL4); 3) delete repeated PLs and SLs (eliminate PL7 that equals PL1 and SL3 that equals SL1); 4) update nodes, PLs, and SLs in the list of CLSs accordingly (substitute `n6` for `n1`, `n4` for `n2`, PL7 for PL1 and SL3 for SL1) and 5) eliminate duplicates in CLSs (CLS2(n1) that equals CLS1(n1)) and rename to avoid numbering gaps (CLS3(`n1`) becomes CLS2(`n1`)).

Node summarization solves the issue of bounding the graphs. Additionally, the graph union operation is used to reconcile different graphs that reach a join point in the CFG. A typical case is the head of a loop, where the graph coming from the last statement in the loop merges with the graph from the previous iteration. Graph union is performed by adding all the information of both graphs into a *working* graph. This graph is then normalized as described in the summarization operation, so that redundancies are removed. The result is a graph that conservatively captures all possibilities for the memory configurations at that point in the analysis.

Then we need to decide wether to enter the loop again or exit it. This is determined by the fixed-point test. If the result of the last iteration, $SG_{loop}^{(i)}$, contains the same information than the previous iteration, $SG_{loop}^{(i-1)}$, then we have reached a fixed-point. In that case, we can leave the loop because we have conservatively registered all possible memory configurations that originate from it. Otherwise, we must keep iterating until the information of the graphs at the head of the loop are equivalent. This state is ensured by the existence of summarization. The graphs cannot grow endlessly nor can we enter a *deadlock* situation.

Node summarization and graph union are necessary for the analysis to work. However, they only provide ways to merge information. Conversely, the materialization operation *separates* information, which is the key to achieve precision in shape analysis, as shown in the following example. Fig. 5(a) depicts a memory configuration for three single-linked lists: lists `x` and `y` share memory locations from the second element onwards, while the `z` list is independent. The shape graph that matches this configuration appears in fig. 5(b). It represents all memory locations that are not directly accessed by stack pointers in a summary node (`n4`). CLSs for `n4` are also shown. They indicate possibilities of connectivity of `n4` with the rest of the nodes in the graph. In particular, CLS1(`n4`) is telling that, when `n4` is reached from `n1`, it is undoubtedly reached from `n2` as well, but not from `n3`. On the other hand, CLS2(`n4`) is telling that when `n4` is reached from `n3`, it is not reached from `n1` or `n2`. Therefore, CLSs are accurately capturing the memory configuration considered for this example. More importantly, since we have all the information about possible connections of links over `n4`, we are able to materialize a new node (`n5`) by traversing the `z` list (`p=z->nxt`), free of links from `n1` and `n2`. No other shape analysis that we know of would be able to discard those *false* links in the materialized node, given such a high level of *compression* for the summary node. This is achieved with very little storage requirements.

6. Related work

In the past decades pointer analysis has attracted a great deal of attention. A lot of studies have focused on stack-pointer analysis while others, more related to our work, have focused on heap-pointer analysis. Both fields require different techniques of analysis. In the context of heap analysis, some authors have proposed approaches that are based on encoding access-paths in path matrices [3] or limited path expressions [5]. Such approaches do not consider a graph representation of the heap. Other authors ([4], [2], [6], [7]) have considered shape

abstraction expressed as graphs, just like us.

Some early shape analysis techniques started with coarse characterization of the shape of the data structures as a matching process with pre-defined shapes, namely tree, DAG (direct acyclic graph) or cycle, like in [3]. In the case of cyclic structures nearly all precision is lost. However, Hwang et al. [5] have achieved some success applying his shape analysis abstraction to dependence detection in programs with cyclic structures traversed in an acyclic fashion.

Sagiv et al. [7] present a graph-based shape analysis framework that sets the foundations for our approach to the shape analysis problem. Their use of abstract interpretation/abstract semantics, along with materialization, were taken in as the basics for the development of our first shape analysis framework [1]. Corbera et. al augmented the analysis precision and introduced the concept of properties in nodes to be able to have separate summary graphs. Later, a powerful loop-carried dependence test was developed over this framework to provide good results for real-life programs that deal with complex pointer-based structures [8]. We propose now a new shape analysis technique based on CLSs that aims to surpass previous works.

7. Conclusions and future work

In this paper we have presented a new strategy for the static analysis of dynamically allocated data structures in pointer-based programs, like those easily found in C or C++. We have presented the key concept of Coexistent Link Sets, that codify in a neat and compact way possibilities of connectivity between memory locations. The sempiternal trade-off between precision and cost, that is so decisive in shape analysis, can be fine-tuned with the help of configurable and extensible properties. We are currently working in a compiler framework that conforms to the CLS description, in order to provide experimental results that can demonstrate the advantages of our approach.

References

[1] F. Corbera, R. Asenjo, and E.L. Zapata. A framework to capture dynamic data structures in pointer-based codes. *Transactions on Parallel and Distributed System*, 15(2):151–166, 2004.

[2] M. Wegman D. Chase and F. Zadek. Analysis of pointers and structures. *In SIGPLAN Conference on Programming Languages Design and Implementation*, pages 296–310, 1990.

[3] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, St. Petersburg, Florida, January 1996.

[4] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. *ACM SIGPLAN Notices*, 1989.

[5] Y. S. Hwang and J. Saltz. Identifying parallelism in programs with cyclic graphs. *Journal of Parallel and Distributed Computing*, 63(3):337–355, 2003.

[6] J. Plevyak, A. Chien, and V. Karamcheti. Analysis of dynamic structures for efficient parallel execution. In *Int'l Workshop on Languages and Compilers for Parallel Computing (LCPC'93)*, 1993.

[7] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, January 1998.

[8] A. Tineo, F. Corbera, A. Navarro, R. Asenjo, and E.L. Zapata. A novel approach for detecting heap-based loop-carried dependences. In *The 2005 International Conference on Parallel Processing (ICCP'05)*, June 2005.

Optimal Tile Size Selection Guided by Analytical Models*

Basilio B. Fraguela[a], Martín G. Carmueja[a], Diego Andrade[a]

[a] Dept. de Electrónica e Sistemas, Universidade da Coruña, Facultade de Informática, Campus de Elviña, 15071 A Coruña, Spain. {basilio,carmueja,dcanosa}@udc.es

As the memory bottleneck problem continues to grow, so does the relevance of the techniques that help improve program locality. A well-known technique in this category is tiling, which decomposes data sets to be used several times in a computation into a series of tiles that are reused before proceeding to process the next tile. This way, capacity misses are avoided. Finding the optimal tile size is a complex task. In this paper we present and compare a series of strategies to search the optimal tile size guided by an analytical model of the whole memory hierarchy and the CPU behavior. Our experiments show that our strategies find better tile sizes than traditional heuristic approaches proposed in the literature while requiring a small compile-time overhead. Iterative compilation can yield better results, but at the expense of very large overheads.

1. Introduction

The success of memory hierarchies in bridging the gap between the processor and the main memory speeds depends on the locality of the data accesses. Such locality can be improved by a number of software optimizations [2]. One of the most effective techniques is tiling [8,10,4,13,7], which combines strip-mining and loop permutation to create small tiles of loop iterations that concentrate the accesses to a given data set, thus maximizing access locality. The tile sizes are chosen so that the associated data sets fit in the cache; which results in a reduction of the capacity misses. The effect of tiling on multiple processors is even more significant, since it not only reduces average data access latency, but also the required memory bandwidth. Still, the fact that caches have a limited associativity and that normally during a portion of the execution of a program several data structures need to be accessed implies that despite the application of this technique, many capacity and conflict misses can take place. Besides, the number of misses may vary widely depending on the chosen tile size, since cache behavior is unstable and very sensitive to small changes on a large number of parameters [10,14].

Most of the approaches proposed so far to choose an optimal tile size are based on relatively simple heuristics [10,4,13,7]. These algorithms have many restrictions. The most important ones are that they adopt a very simplified view of the cache behavior, they do not consider the additional CPU time required to manage the tiles, they are restricted to a single cache level, and they only consider accesses to a single data structure. As a result, very often the tiles proposed by these techniques are far from being optimal. Iterative compilation techniques [9] to explore the solution space yield better results. Unfortunately these techniques are too costly to be applied repetitively and/or in big applications except in very particular cases.

An optimal tile size search technique introduced in [5] lies in exploring the solution space guided by a precise analytical model that considers both the CPU and the whole memory hierarchy cost of a tile size. This paper compared the model predictions for a fixed set of tile sizes that were divisors (or values nearby) of the sizes of the loops in the considered nests and chose the one with the lower cost.

*This work has been supported in part by the Ministry of Science and Technology of Spain under contract TIN2004-07797-C02-02, and by the Xunta de Galicia under contract PGIDIT03-TIC10502PR.

Later, [15] applied similar ideas with other models using a genetic search algorithm. Both works lacked a study of which of the possible search strategies would yield the best results. In this paper we compare both search techniques as well as a novel hybrid one we propose here which turns out to yield the best results. We also compare the results of search guided by models with those of the traditional heuristic approaches and iterative compilation.

The rest of the paper is organized as follows. An introduction to our model of the memory hierarchy is provided in Sect. 2. Our approach to estimate the cost associated to a tile size is explained in Sect. 3. Then, Sect. 4 presents the optimal tile size search strategies that we consider. The experiments in Sect. 5 shows the relative advantages of the different tile size selection techniques. Section 6 contains a review of related work. Finally, Sect. 7 is devoted to our conclusions.

2. The Probabilistic Miss Equations (PME) Model

The PME model [5] provides fast and accurate predictions of the memory behavior of codes with regular access patterns in direct-mapped or set-associative caches with a LRU replacement policy. The model is based on the idea that cache misses take place the first time a memory line is accessed (compulsory miss) and each time a new access does not find the line in the cache because it has been replaced since the previous time it was accessed (interference miss). This way, the model generates a formula for each reference and loop that encloses it that classifies the accesses of the reference within the loop according to their reuse distance, this is, the portion of code executed since the latest access to each line accessed by the reference. The reuse distances are measured in terms of loop iterations and they have an associated miss probability that corresponds to the impact on the cache of the footprint of the data accessed during the execution of the code within the reuse distance. In particular, if K is the degree of associativity, the miss probability is given by the ratio of cache sets that have received K or more lines during the execution of the code within the reuse distance. The reason is that if each cache set holds K lines, a line whose behavior is being observed will have been replaced if and only if K or more different lines mapped to its cache set have been accessed since the previous access to the line. The formula estimates the number of misses generated by the reference in the loop by adding the number of accesses with each given reuse distance weighted by their associated miss probability.

The accesses that cannot exploit reuse in a loop are compulsory misses from the point of view of that loop. Still, they may enjoy reuse in outer loops, so they must be carried outwards to estimate their potential miss probability. This is why the PME model begins the construction of its probabilistic formulas in the innermost loop that contains each reference and proceeds outwards. The formulas are built recursively, with the formula for each loop level being built in terms of the formula obtained for the immediately inner level. In each nesting level the set of accesses whose reuse distances correspond to iterations of that loop are detected, and the associated miss probabilities are estimated. Those accesses for which no reuse distance has been found when the outermost loop is reached correspond to the absolute compulsory misses, whose miss probability is one.

The usage of miss probabilities, generated by estimators provided by the model of the impact on the cache of the accesses to data regions during the reuse distances, gives place to the model's probabilistic nature, which distinguishes it from all the other cache models in the bibliography.

3. Computer Modeling

The PME model provides accurate estimations of the behavior of a cache, but current computers have a memory hierarchy with several levels of caches, and the CPU plays of course an essential role

in the system performance too. The first problem is solved by extending the PME model to consider a hierarchy of cache levels. Each level i is characterized by a total cache size C_{s_i}, a line size L_{s_i}, and an associativity K_i. Given these parameters and an arbitrary code, the model can predict the number of misses generated in each level of memory hierarchy. The property of inclusion that multilevel memory hierarchies fulfill allows the independent modeling of the different levels, as each access that were a hit in a given cache level, should also be a hit in the lower levels as a consequence of this property. Still, the caches in the different levels do not experience the same pattern of accesses, since accesses are filtered by the successive levels as they proceed down the hierarchy. This fact can generate some deviations with respect to this ideal behavior, but they are minimal and so they affect little the accuracy of the estimations obtained following this approach.

Since we aim to predict the performance of the whole memory hierarchy, misses in the different levels receive different weights W_i that are given by the miss penalty in cache level i measured in processor cycles. This way, the cost of the execution of loop nest L using the set of tile sizes T in a system with N cache levels can be estimated as

$$\text{MemCost}(L, T, H) = \sum_{i=1}^{N} W_i M(C_{s_i}, L_{s_i}, K_i, L, T) \tag{1}$$

where function M yields the number of misses estimated by the model for loop nest L for a given cache level and set of tile sizes; and H stands for the memory hierarchy represented as a set of tuples $H = \{(C_{s_1}, L_{s_1}, K_1, W_1), \ldots, (C_{s_N}, L_{s_N}, K_N, W_N)\}$.

We use the Delphi [3] CPU model in order to estimate the number of cycles $\text{CPUCost}(L, T, C)$ that code L spends in the CPU C depending on the tile sizes T. This model simply counts the number of operations of each type found within the code and it adds them weighting them according to their respective latencies. Since current processors are superscalars, Delphi uses some simple heuristics to take into account the overlapped execution of instructions so as not to overestimate the CPU time.

This way our final estimation of the cost of executing loop nest L with the set of tile sizes T in a computer with a CPU C and a memory hierarchy H is given by

$$\text{Cost}(L, T, C, H) = \text{MemCost}(L, T, H) + \text{CPUCost}(L, T, C) \tag{2}$$

The Delphi and the PME models, as well as our optimal tile size search module are implemented in the Polaris platform [1], which we use to analyze the codes and generate the optimized versions with the tile sizes chosen by our module.

4. Search Strategies

Our approach to find the tile sizes that minimize the execution time of a loop nest L lies in exploring the solution space (the combinations of possible tile sizes) guided by our analytical model. Such exploration can be performed using several strategies. In our experiments we have explored the viability of two completely different strategies: the search on divisors, an ad-hoc algorithm specifically designed for this problem that searches only in a set of predefined values, and a genetic algorithm, which is a general search approach applicable to any solution space. Then, we have developed a hybrid strategy that combines them.

4.1. Search on Divisors

In each loop nest in which tiling has been applied, this algorithm searches a well-defined subset of all the combinations of possible tile sizes. Concretely, for each tiled loop it chooses the initial

Table 1
Cache and TLB parameters in the architectures used (sizes in Bytes)

Architecture	L1 Parameters $(C_{s_1}, L_{s_1}, K_1, W_1)$	L2 Parameters $(C_{s_2}, L_{s_2}, K_2, W_2)$	L3 Parameters $(C_{s_3}, L_{s_3}, K_3, W_3)$	TLB Parameters $(C_{s_4}, L_{s_4}, K_4, W_4)$
Pentium 4	(8K,64,4,24)	(512K,128,8,150)	-	(256K,4K,64,30)
Itanium 2	Irrelevant	(256K,128,8,24)	(6MB,128,24,120)	(8MB,64K,128,25)

tile size specified by the program, if any, and the series of values $T_i = \lceil N/i \rceil$, where N is the total loop size. The intention of choosing divisors of the loop sizes is to maximize the work made in each iteration of the loops that traverse the tiles, and to simplify the control conditions of the loops. In our experiments we used $0 < i \leq 128$, and whenever two values of T_i differed in less than three units, one of them was discarded. This algorithm, applied in [5], searches the best combination of values chosen from these sets for the different tiles in a loop nest. This approach explores quickly the solution space paying more attention to small tile sizes.

4.2. Genetic Algorithm

Genetic algorithms are stochastic methods applicable to problems for which no specific resolution method is available. These algorithms simulate the genetic and natural selection processes. Namely, an initial set or *population* of candidate solutions is encoded as bit strings arbitrarily codified. These strings are modified and recombined to produce new solutions. The solutions are evaluated by means of a *fitness function* that chooses the best ones and promotes them as the base for a new *generation* of solutions. Crossover and mutation are applied on minimal units of information called *genes* in order to recombine and alter the strings. The process of generation and selection of solutions is applied repetitively for several generations till certain convergence criteria are met. The adaptability of these algorithms, a good coverage of the solution space and their inherent parallelism make them suitable for the search of the optimal tile size.

4.3. Hybrid Algorithm

Both the search on divisors and the genetic algorithm have interesting properties to solve our problem. It is possible to combine their advantages following a hybrid strategy. The hybrid algorithm we propose has two phases. In the first step, an approach to the solution is found using the search on divisors. In the second phase, the result is refined using a genetic algorithm. The n best solutions found in the first phase constitute the initial population for the second phase. This population is completed using versions with noise of elements randomly chosen among the n first ones.

5. Evaluation

For each search strategy we implemented two versions in our tool: one based on the exploration of the solution space guided by the PME analytical model, and another one based on iterative compilation [9]. We also implemented two of the most popular traditional techniques based on heuristics in order to compare them with our strategies. The first one, which we call *lrw* [10], chooses the biggest square block that does not collide with itself in the cache. The second technique, which we name *euc*, was developed in [13] by extending the work in [4], based on the Euclidean GCD algorithm.

We run our experiments in two popular platforms: a Pentium 4 at 2 GHz with g77 3.3, which is representative of the most widely extended architecture nowadays, and an Itanium 2 at 1.5 GHz with a 6MB third level cache and the HP F90 2.7.3 compiler. The `O3` optimization level, and flags to preclude the compiler for applying additional loop transformations that would distort the experiments were used to compile the codes in both systems. Table 1 shows the configuration of the TLB

Table 2
Description of the kernels and sizes used for the experiments, where $i = 0, 1, \ldots, 19$.

Kernel	Description	Pentium 4 sizes	Itanium 2 sizes
MXM	Matrix product (IJK)	$300 + 50 \times i$	$1000 + 50 \times i$
MV	Matrix-vector product	$300 + 50 \times i$	$2000 + 50 \times i$
TRANSP	Bidimensional matrix transposition	$1700 + 50 \times i$	$4000 + 50 \times i$
VPENTA	Inversion of three pentadiagonals	$800 + 50 \times i$	$2000 + 50 \times i$

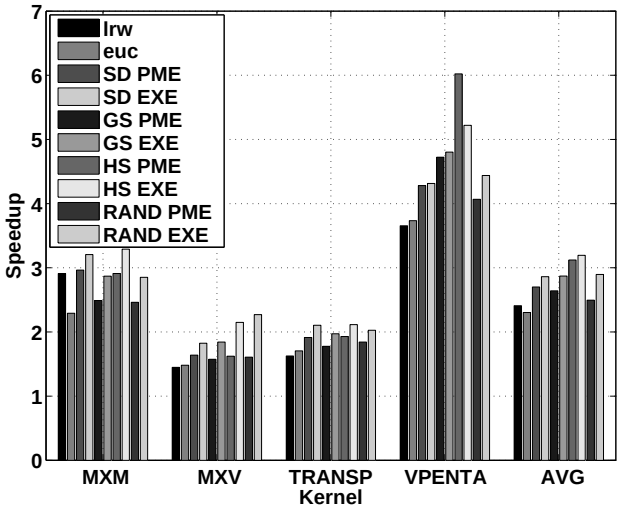


Figure 1. Tiling speedup in the Pentium 4 as a function of the strategy followed to choose the tile sizes

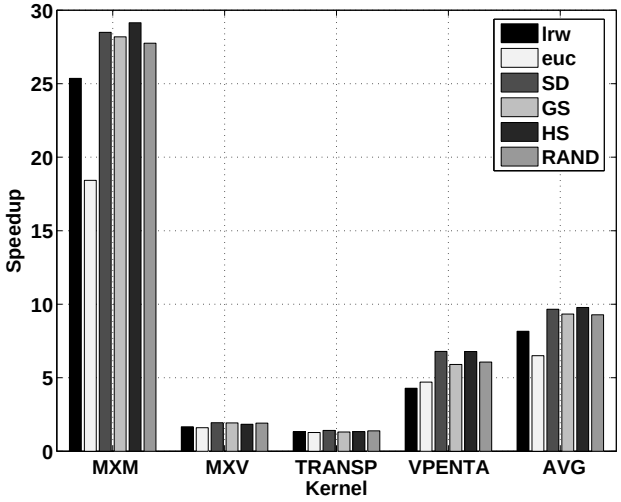


Figure 2. Tiling speedup in the Itanium 2 as a function of the strategy followed to choose the tile sizes

and the cache levels of both systems. The TLB acts as a level of the memory hierarchy from the point of view of the execution time, and this is how it is regarded by the analytical model. In fact it can be characterized with the same parameters as a cache, the page size being the line size, and the product of the page size by the number of entries of the TLB being the total size of the level. Let us notice that the Pentium 4 has only two level of caches, while the first level cache of the Itanium 2 is irrelevant for our experiments, since it does not store floating point data; which is the kind of data we use in our experiments. As for the CPU parameters, the most relevant one from the point of view of the tile size selection is the penalty associated to misspredicted branches, which is 20 cycles in the Pentium 4 architecture and 8 for the Itanium 2.

We used for our experiments four representative kernels taken from [15]. Table 2 describes them briefly and shows the sizes considered for our experiments in both platforms. The sizes used in the Itanium 2 are much larger than those used in the Pentium 4, since while the first level of cache to consider in the Itanium 2 has 256 KBytes, the first level cache of the Pentium 4 has only 8 KBytes. Something similar happens when we compare the size of their corresponding next level caches and the TLBs.

Figures 1 and 2 show the speedups achieved by each tiled algorithm with respect to its non-tiled version depending on the strategy followed to choose the tile size on the Pentium 4 and the Itanium 2 architectures, respectively. The last set of bars, labeled AVG, represents the arithmetic mean of

the speedup achieved over the four kernels for each strategy. Columns *lrw* and *euc* correspond to the traditional techniques described in [10] and [13] respectively, as explained before. Bars SD, GS, HS and RAND correspond to the Search on Divisors (Sect. 4.1), the pure Genetic Algorithm Search (Sect. 4.2), the Hybrid Search (Sect. 4.3) and a Random Search, respectively. The parameters used for the genetic algorithms in GS and HS are very similar to those used in [15] and [9]. As for the hybrid search, its genetic search started with a population formed by the 30 best tiles found by the search on divisors. The random search tried 184 randomly generated tile sizes, which is a value similar to the average number of tiles considered by the genetic algorithm search. In the Pentium 4 architecture we tested each search strategy by guiding it either by the PME analytical model static predictions or by means of real executions (iterative compilation). Both columns are labeled PME and EXE in Fig. 1 respectively. We could only run experiments using the static predictions in our Itanium 2 because of its limited availability and the large amount of time that iterative compilation requires.

We can see that while the tiles chosen by the genetic search (GS) and the random search (RAND) sometimes performed worse than those of the traditional approaches, the search on divisors (SD) and the hybrid search (HS) always generated results at least as good as those of the heuristics. As expected, iterative compilation generates almost always better results than the static model, but in general the difference is small: on average the tiles chosen by the iterative compilation are 8.3% faster than those chosen by the static search, and just 5.7% if we exclude the blind random search. Interestingly, HS PME chooses a much better tile than HS EXE in VPENTA. This may be due to the random factor in any genetic algorithm. The average speedup of the static SD, GS, HS and RAND PME searches over the best traditional approach (*lrw*) is 12%, 9.6%, 29.5% and 3.53% in the Pentium 4, respectively; and 18.4%, 14.4%, 19.8% and 13.8% in the Itanium 2, respectively. This way, the novel hybrid search we propose in this paper seems to be the best overall strategy, followed by the search on divisors.

Tile selection guided by the PME analytical model is completely feasible for current compilers, given that the average time to choose the tile sizes was about 0.82 seconds per code in the Pentium 4 and 2.15 seconds in the Itanium 2, with maximum times below 7 seconds in both architectures. In contrast, iterative compilation times are usually in the order of the thousands of seconds in the Pentium 4, with an average search time of 2709 seconds, and a maximum time of 20440 seconds.

6. Related Work

Traditional approaches to choose the optimal tile size [10,4,13,7] have many limitations: they disregard the CPU time, they focus on a single level of the memory hierarchy and they rely on simple heuristics that dismiss important cache parameters like the associativity, that only consider the reuse in a single array, and that pay little or no attention to the interactions in the cache among several data structures, which could include several tiles generated by the dimension partitioning implied by tiling. For example, the fact that traditional approaches only consider a single tile while our general analytical model considers arbitrary inter-tile interactions is one of the reasons why the speedups achieved by our approach with respect to the traditional ones is much more noticeable in VPENTA than in the other kernels: this code has two tiled loop nests in which tiling defines at least three different tiles that interact in the cache.

The importance of taking into account the interactions in several levels of the memory hierarchy and using global metrics rather than focusing on local cost functions when choosing tile sizes has been proved in [11]. Still, this work does not propose any general framework for this purpose: it is based on a specific model with many simplifications for a particular code. A framework for the op-

timal multi-level orthogonal tiling problem for fully permutable, perfectly nested, rectangular loops that are compute bound, i.e., in which the amount of computation is at least one order greater than the amount of memory operations, is presented in [12]. This high-level model does not consider may factors like cache associativity, caches hits/misses, etc., but the authors suggest applying simple heuristics to include them in the model. Interestingly, the paper shows that the model tracks approximately the execution time of simple loops, but there are no measurements on the quality of the tiles the framework is supposed to choose. Multi-level semi-oblique tiling, which is more general than the orthogonal tiling considered by our paper and most works, has been studied by [16] and [6] in the context of multiprocessors. The applicability of [16] is restricted to a class of iterative stencil calculations, and it does not address the problem of tile size optimization. The approach in [6] is much more general, but it only handles perfectly nested loops and it does not provide any model for the memory memory behavior; rather it focuses on improving parallelism via minimization of the longest path of dependent tiles in the iteration space.

Iterative compilation [9] is based on the real execution on the machine, so it is the most reliable strategy to choose optimal tile sizes. Still, it is only applicable when long times can be devoted to the optimization process, and in our experiments the tiles it chooses are only fractionally better on average than those chosen by an exploration of the solution space guided by a good analytical model. In fact, the analytical model can even choose better tiles than the iterative compilation, since random factors may affect the search, as we have seen in our validation. The hybrid search strategy proposed in this paper has proved to be of interest for both static and iterative tile selection approaches.

Tile and pad factors selection guided by analytical models using a genetic algorithm search is explored in [15]. The relative speedups over *lrw* they achieve on a Pentium 4 with the same characteristics as the one used in our experiments (8%) are similar to those measured in our validation when we use the same kind of search (9.6%), while the times they require to drive the optimization process are several times longer than ours. A motivation for our work was to compare the genetic algorithm search approach applied in [15] with our search on divisors [5]. The latter yields better results in our experiments, but the optimal search strategy turns out to be the mixture of both strategies in the hybrid approach we present in Sect. 4.3.

7. Conclusions

In this paper we propose a new search strategy for the optimal tile size that combines the search on divisors of the loop sizes with a pure genetic algorithm approach. We also compare the traditional heuristic-based approaches to choose tile sizes with different algorithms that search the solution space guided by either estimations of an analytical model that predicts the performance of the computer, or measurements of actual executions (iterative compilation). Our experiments show that while iterative compilation requires three to five orders of magnitude more time than analytical model based search, on average it only delivers relatively small performance improvements over search guided by analytical models and in fact it can (seldom) deliver worse results. The search based on the PME analytical model estimations is suitable to be used in production compilers, since it typically requires one or two seconds; and never more than 7 seconds in our experiments. As for the quality of the search, the algorithms based on the exploration of the solution space almost always generate better results than the heuristical approaches, with our novel hybrid search being the best strategy, followed by the search on divisors. Both approaches are always better than the traditional heuristics.

Future work includes extending our tool to apply more optimizations guided by our analytical model and generalizing the PME model to cope with codes with irregular access patterns.

Acknowledgments

We want to acknowledge the Centro de Supercomputación de Galicia (CESGA) for the usage of its supercomputers to get the data related to the Itanium 2 architecture.

References

[1] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Parallel Programming with Polaris. *IEEE Computer*, 29(12):78–82, 1996.

[2] S. Carr, K. S. McKinley, and C-W. Tseng. Compiler Optimizations for Improving Data Locality. In *Proc. of the Sixth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI), Computer Architecture News*, volume 22, pages 252–262, Boston, MA, Oct. 1994. ACM SIGARCH/SIGOPS/SIGPLAN.

[3] G.C. Cascaval. *Compile-time Performance Prediction of Scientific Programs*. PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Aug 2000.

[4] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *Proc. of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI'95)*, pages 279–290, June 1995.

[5] B. B. Fraguela, R. Doallo, and E. L. Zapata. Probabilistic Miss Equations: Evaluating Memory Hierarchy Performance. *IEEE Transactions on Computers*, 52(3):321–336, March 2003.

[6] K. Hogstedt, L. Carter, and J. Ferrante. On the parallel execution time of tiled loops. *IEEE Trans. Parallel Distrib. Syst.*, 14(3):307–321, 2003.

[7] C-H. Hsu and U. Kremer. A quantitative analysis of tile size selection algorithms. *Journal of Supercomputing*, 27(3):279–294, 2004.

[8] F. Irigoin and R. Triolet. Supernode partitioning. In *Proc. of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 319–329, San Diego, California, 1998.

[9] T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *Proc. Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'00)*, pages 237–248, 2000.

[10] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *Proc. of the Fourth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 63–74, Santa Clara, California, Apr 1991. ACM SIGARCH, SIGPLAN, SIGOPS, and the IEEE Computer Society.

[11] N. Mitchell, K. Hogstedt, L. Carter, and J. Ferrante. Quantifying the multi-level nature of tiling interactions. *Int. J. Parallel Programming*, 26(6):641–670, 1998.

[12] L. Renganarayana and S. Rajopadhye. A geometric programming framework for optimal multi-level tiling. In *Proc. 2004 ACM/IEEE Conference on Supercomputing (SC'04)*, page 18, Washington, DC, USA, Nov. 2004. IEEE Computer Society.

[13] G. Rivera and C.-W. Tseng. A Comparison of Compiler Tiling Algorithms. In *Proc. of 8th Int'l. Conf. on Compiler Construction*, volume 1575 of *Lecture Notes in Computer Science*, pages 168–182, 1999.

[14] O. Temam, C. Fricker, and W. Jalby. Cache Interference Phenomena. In *Proc. Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 261–271. ACM Press, May 1994.

[15] X. Vera, J. Abella, A. Gonzalez, and J. Llosa. Optimizing program locality through CMEs and GAs. In *Proc. 12th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'03)*, pages 68–78, New Orleans, Louisiana, October 2003.

[16] David Wonnacott. Time skewing for parallel computers. In *Proc. of the 12th Intll. Workshop on Languages and Compilers for Parallel Computing (LCPC '99), LNCS vol. 1863*, pages 477–480, London, UK, 2000. Springer-Verlag.

Pack Transposition: Enhancing Superword Level Parallelism Exploitation*

C. Tenllado[a], L. Piñuel[a], M. Prieto[a], F. Catthoor[b]

[a]Dpto. de Arquitectura de Computadores y Automática
Universidad Complutense, 28040 Madrid, Spain
e-mail: {tenllado, lpinuel, mpmatias}@dacya.ucm.es

[b]IMEC
Kapeldreef 75, B-3001 Leuven, Belgium
e-mail: catthoor@imec.be

Current compilers do not allow for an efficient exploitation of the *Superword Level Parallelism* (SLP) available in many image processing applications. In this paper we present a modified version of the SLP compiler algorithm introduced by Larsen, that overcomes some of these problems. As motivating examples we have considered two kernels extracted from multimedia application, in which state-of-the-art compilers fail to exploit the available fine-grain data parallelism. Our methodology manages to extract the available SLP and achieves consistent speedups on both an Intel Pentium 4 and an Apple G4-based platform.

1. Introduction

Contemporary computer applications are multimedia-rich, involving significant amounts of audio, video and image processing. This trend has resulted in a variety of new ISA extensions, usually denoted as *Multimedia Extensions*, to practically all general-purpose microprocessors, including IBM-Motorola's AltiVec [3] for PowerPC, and Intel's MMX, SSE1, SSE2 and SSE3 [4,5] for Pentium. While different processors vary in the type and number of multimedia instructions offered, at the core of each is a set of short SIMD style operations. These instructions boost the performance of media applications operating concurrently on data that are packed in advance in a single wide (vector) register.

Unfortunately, these extensions do not provide yet transparent performance improvements. Despite the early success of automatic vectorization for traditional vector supercomputers, state-of-the-art vectorizing compilers for multimedia extensions have yet to demonstrate their effectiveness. In fact, applications developers usually turn to explicitly hand-tune their codes using in-line assembly, intrinsic functions or specialized library routines.

Classic approaches for automatic vectorization, such as the Allen-Kennedy algorithm [6], are based on the theory of data-dependence analysis, which was developed during the 70's and the 80's for array-based Fortran programs from the scientific computing domain. Dependence analysis is used to detect loop statements that could be executed in parallel without violating the semantics of the program (vector loops), and loop transformations, such as loop interchange or loop fission, were developed to increase such occurrences.

The similarity with vector processors has prompted the adaptation of classic approaches to compile for multimedia extensions. However, differences in both the target architecture (especially in the memory system) and the target domain make this adaptation difficult. General-purpose architectures

*This work has been supported by the Spanish goverment research contract TIC 2002-750 and the Hipeac European Network of Excellence

only support efficiently accesses to adjacent memory addresses, and only on vector length aligned boundaries. Although *Media Extensions* usually provide mechanisms to reorganize data elements in vector registers to deal with such situations (packing, unpacking and special shuffle instructions), they are not easy to use, and incur considerable penalties. On the other hand, traditional loop-based vectorization has been focused on statically analyzable codes, where little or no dynamic behavior is present, whereas multimedia codes are no longer static and make extensive use of pointers, making data dependence analysis more complex.

Our research is based on the *Superword Level Parallelism* compiler, an alternative approach introduced by Larsen et al. in [8]. It is focused on packing isomorphic instructions from the same basic block to vector instructions, and can be seen as a restricted form of *Instruction Level Parallelism* (ILP). We have extended the Larsen's compiler with additional transformations aimed at extracting the available SLP in a more efficient way.

The rest of the paper is organized as follows. In Section 2 we describe a motivating example to illustrate the problems under scope. Section 3 summarizes the original SLP compiler. Section 4 introduces our proposed enhancements, analyzing in Section 5 their efficiency on two different architectures, a Power-PC platform and an Intel Pentium4. Finally the paper ends with some conclusions.

2. Motivating example

A motivating example extracted from 2D image processing [2] is illustrated in Figures 1 and 2. The same algorithm is applied first to the image rows and then to the image columns. The array is scanned row by row in both cases due to some locality optimizations performed in early steps of the compilation process.

```
for i=0 to N-1
   for j=0 to N-3
      A[i,j+1] = α*A[i,j]+β*A[i,j+2];
```

Figure 1. A simple 1D algorithm applied on rows.

```
for i=0 to N-3
   for j=0 to N-1
      A[i+1,j] = α*A[i,j]+β*A[i+2,j];
```

Figure 2. A simple 1D algorithm applied on columns.

In the situation described by Figure 2, vector style parallelism is easily exploited by state-of-the-art approaches, since all the dependencies are carried out by the external loop [10,8]. However, they become inefficient when the dependencies are carried out by the inner loop (Figure 1). A classic vector compiler would apply loop interchanging [10], but this is not appropriate in our context since, it produces an access pattern with poor spatial locality and elements to be packed are not stored in adjacent memory addresses. The larsen's compiler would unroll the inner loop, which allows for a partial vectorization.

Is it possible to extract more SLP in cases similar to the one described in Figure 1? In this paper we try to answer this question. Basically, we propose some modifications to the SLP compiler algorithm that allow us to efficiently exploit the vector parallelism available in the outer loop.

3. Overview of the SLP compiler

Before describing the SLP compiler, it is convenient to remind some definitions introduced in [8]:

Definition 3.1 *A Pack is an n-tuple,* $\langle s_1, s_2, s_3, \cdots, s_n \rangle$*, where* $s_1, s_2, s_3, \cdots, s_n$ *are independent isomorphic statements in a basic block.*

Definition 3.2 *A PackSet is a set of Packs.*

Definition 3.3 *A Pair is a Pack of size two, where the first statement is considered the left statement, and the second statement is considered the right element.*

Definition 3.4 *The SuperWord Size (*sws*), is the maximum number of data elements that can be packed in a short vector register on the target platform.*

The SLP compiler extracts parallelism from the innermost basic block in a loop nest. It is based on a pre-processing, in which the innermost loops are unrolled by a factor equal to the sws. This unrolling constructs a basic block with several consecutive instances of the same statement. If vector parallelism could be extracted from the original loop, it can now be extracted from the basic block.

The core of the SLP compiler is applied later on a three-address representation of the code. It is subdivided into four phases: *Adjacent Memory Identication*, *PackSet Extension*, *Combination* and *Scheduling*, which are described bellow.

In the *Adjacent Memory Identication* stage, the basic block is scanned searching for *Pairs* with adjacent memory references, which are grouped together forming the initial *PackSet*. Adjacency is determined using both alignment information and array analysis. No *Pairs* are formed that cross alignment boundaries.

Statements can belong simultaneously to two *Pairs* as long as they occupy the left and right positions in the two *Pairs* respectively. This allows the *Combination* stage to easily merge groups into larger clusters. A simple example of the process is described in Figure 3.

More *Pairs* are added to the *PackSet* in the next stage. The compiler does it following the *use-def* and *def-use* chains of the *Pairs* that are currently in the *PackSet*. In this way the new members will consume superwords already formed or will provide the ones needed for an existing *Pair*. In all cases alignment consistency is checked.

Once all the possible candidates have been discovered, the *combination* stage is started. Here two *Pairs* are combined if the right statement of the first *Pair* is the same than the left statement of the other (Figure 3). The combined *Pairs* form a *Pack*. This process continues till no further combination is possible. The alignment consistency guaranties that the *Packs* formed will never cross alignment boundaries and that its size will not exceed the sws.

Finally the *PackSet* contains potential *Packs* of statements that can be executed in parallel using SIMD extensions. Nevertheless, it could happen that executing two groups of statements in parallel produces a dependency violation. A dependency cycle among *Packs* indicates an invalid set of groups, being necessary to remove at least one of the *Packs*.

After *Scheduling*, every *Pack* in the *PackSet* corresponds to a SIMD instruction, and possibly additional pack/unpack instructions. Refer to the original paper [8] for more details.

The weakest points of the SLP compiler are:

- It cannot efficiently extract SLP when dependencies are carried by the inner loop.
- The statement Packing is not steered. It could potentially be enhanced if the packing process is performed according to some information extracted from dependence analysis.
- It does not consider the chance of combining superwords to obtain new seeds for the *Packing* process. The number of *Packs* finally created could be larger taking this into account.

Some of these points are covered in our methodology for the kind of algorithms under scope.

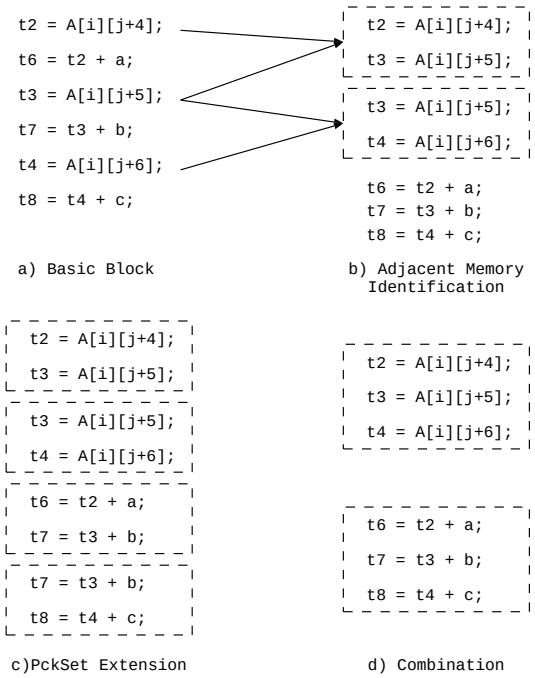


Figure 3. Simple example to illustrate the original SLP methodology.

4. Methodology description

Our methodology, which we have denoted as PT-SLP: *Pack Transposition* SLP, targets two level loop nests that operate on two dimensional arrays with the following constraints:

- The inner loop iterates over the lowest dimension of the arrays (assuming a row-major layout).
- All the dependences in the loop nest are carried by the inner loop. As dependences we consider flow, anti, output and input dependences.
- 2D structures represented by 1D arrays are accessed by affine functions

For this kind of algorithms, the loop unrolling performed by Larsen's compiler is not enough to uncover the vector parallelism. Alternatively, we propose to use first an *unroll-and-jam* transformation in order to uncover the vector parallelism available in the external loop. Figure 4 illustrates this transformation for the example in Figure 1. It consists in unrolling the external loop and fusing the resultant instances of the inner loop. The unrolling factor, as in the original SLP compiler algorithm, corresponds to the sws.

To improve SLP exploitation, we have also added to the SLP compiler algorithm both *loop peeling* and *dynamic alignment detection* techniques [9,7,1].

One of the keys to success of the SLP algorithm is its ability to seed the initial *PackSet* with pairs of statements that imply accesses to adjacent memory locations. This translates into a reduction in the number of load instructions and enables the compiler to find vector candidates that are already packed in memory. These adjacent memory accesses can also be found in the basic block generated by the process described in Figure 4. Thus we do not modify the first phase.

However, we modify the original ordering performing the *combination* stage just after the *adjacent memory identification*. In this way, at the end of the *combination* phase the compiler has a

```
for i=0 to N-1
   for j=0 to N-3
      A[i,j+1] = α*A[i,j]+β*A[i,j+2];
           a) Original code

for i=0 to N-1 by 2
   for j=0 to N-3
      A[i,j+1] = α*A[i,j]+β*A[i,j+2];
   for j=0 to N-3
      A[i+1,j+1] = α*A[i+1,j]+β*A[i+1,j+2];
           b) External loop unroling

for i=0 to N-1 by 2
   for j=0 to N-3
      A[i,j+1]   = α*A[i,j]  +β*A[i,j+2];
      A[i+1,j+1] = α*A[i+1,j]+β*A[i+1,j+2];
           c) Jam

for i=0 to N-1 by 2
   for j=0 to N-3 by 2
      A[i,j+1]   = α*A[i,j]    + β*A[i,j+2];
      A[i+1,j+1] = α*A[i+1,j]  + β*A[i+1,j+2];
      A[i,j+2]   = α*A[i,j+1]  + β*A[i,j+3];
      A[i+1,j+2] = α*A[i+1,j+1]+ β*A[i+1,j+3];
           d) Inner loop unrolling
```

Figure 4. *Unroll-and-jam* plus inner loop unrolling for the example described in Figure 1 ($sws = 2$).

PackSet($\mathcal{P}_0$) that only contains *Packs* of statements that access adjacent memory locations. In the code generation phase, each of these *Packs* can be translated to a vector load. For the same reasons as in the original SLP compiler algorithm, it is guarantied that these *Packs* do not exceed the sws and will not cross alignment boundaries.

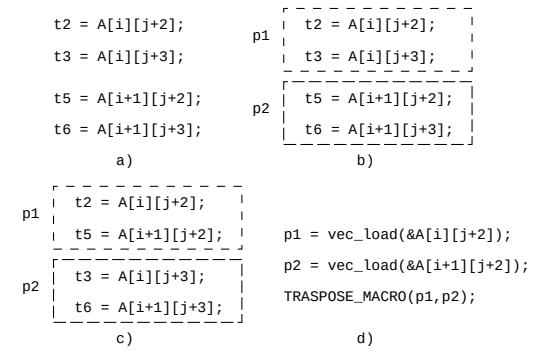


Figure 5. An example of *Pack Transposition* phase and the respective vector code generation for the example in Figure 2 ($sws = 2$).

As a result of the *unroll-and-jam* transformation followed by the inner loop unrolling, we have in the basic block sets with sws equivalent *Packs*. These sets contain statements that perform the same computations on different rows of the arrays (Figure 5b). We will refer to a set of these equivalent *Packs* as a *Group*.

Each *Group* can be seen as a $sws \times sws$ matrix. We introduce a new *Pack Transposition* phase, in order to pack together statements that operate on data elements from different rows. It consists in transposing each of the *Groups* in the *PackSet*. The process is described in Figure 5 for the final

basic block in Figure 4. A new *PackSet* ($\mathcal{P}_1$) is then constructed.

In the code generation phase, this *Pack Transposition* translates into a set of shuffling operations that transform each of the superwords processed by a *Group* in $\mathcal{P}_0$ to the corresponding superwords consumed by the new *Packs* in $\mathcal{P}_1$ (Figure 5d). This process can be done if multidimensional arrays are padded in the lowest dimension, which guaranties constant alignment among rows [8]. As we will show, the shuffling overhead is by far compensated by an increase in the number of short vector instructions finally generated. This new stage is an example of how the packing of elements can be steered according to a dependence analysis.

Finally, the new *PackSet* ($\mathcal{P}_1$) is considered as seed for the *PackSet Extension* and *Scheduling* phases of the original SLP compiler algorithm, with one slight difference, the *PackSet Extension* phase has to work on *Packs* not on *Pairs* of statements.

5. Performance results

As computing platform to evaluate our methodology we have used a 2.4GHz Pentium 4 (768MB 233MHz DDR, 512kB L2) and a 1.42GHz G4 (1GB 167MHz DDR, 512kB L2). As back-ends, we have used the Intel C/C++ compiler (v8.1) and the Altivec Interface of the gcc-3.3. In all cases we have used single precision floating-point as default datatype, which implies sws=4 in both platforms.

In the following subsections we analyzed in detail the performance achieved on our motivating example (the synthetic kernel introduced in Figure 1) and a real kernel extracted from the JPEG2000 (namely, the horizontal filtering of its intra-component transform). In both cases, the automatic vectorization capabilities of the Intel compiler fail to extract the available parallelism.

5.1. Synthetic Kernel

In this example, the original SLP compiler also manages to generate some SIMD code. Its performance is shown in Tables 1 and 2. Given that loop unrolling improve performance by itself, we have analyzed separately this contribution to isolate the benefits of the SLP extraction. As can be noticed, results are qualitative different in both platforms. On the G4, the benefits of the Larsen compiler are negligible and all the performance improvements are achieved by the loop unrolling. In contrast, on the Intel Pentium this loop transformation degrades the performance, although the vectorial loads introduced by the SLP compiler counteract this effect, achieving moderate speedups (between 7% and 12%) over the original code.

Tables 3 and 4 illustrate the benefits achieved by our methodology. As expected, the *unroll-and-jam* transformation improves performance (UJ columns). The other columns show the speedups achieved by SLP and our methodology (denoted as PT-SLP: *Pack Transposition SLP*) over this optimized scalar code. As can be noticed, *unroll-and-jam* does not fit well with the original SLP compiler (SLP+UJ columns), especially on the Intel platform , in which their combination translates into noticeable performance slowdowns.

Our methodology always manages to achieve additional speedups: the overheads introduced in the *Pack Transposition* phase are by far compensated by the additional SLP extracted. Results are qualitative similar on both platforms (i.e. our approach is more robust that the original SLP compiler). For small problem sizes the speedups are close to the ideal values, whereas for the largest problems, in which this benchmark becomes memory bounded, the benefits of the SLP extraction decrease. This behavior highlights the strong influence of the memory hierarchy in the SLP exploitation.

5.2. Intra-component Transform

In this real application, the SLP compiler fails to extract the available SLP, and hence Tables 5 and 6 only show the benefits of the *unroll-and-jam* transformation and the overall speedups achieved by our methodology. Given that *unroll-and-jam* can degrade performance (especially on the Intel Pentium) in this case, the speedups of our methodology refers to original scalar code.

The qualitative results are again similar in both platforms, and the speedup decreases with the problem size. For large problems, this application also becomes memory bounded and the speedups decrease. Quantitatively, gains are outstanding on the G4 platform, in which the speedups for the small problem sizes match the ideal values.

6. Conclusions

In this paper we have introduced a novel compiling methodology, denoted as PT-SLP, that efficiently extracts SLP in some applications in which current compilers fail to generate efficient short vector code.

Our methodology clearly outperforms the original SLP compiler and also exhibits a more robust behavior. Speedups are qualitatively similar on the target architectures, although quantitative results has been much better on the G4 platform. The speedups are higher and close to the ideal values for small problem sizes (around 4 on the G4 and 3 on the Pentium4), and they decrease for large problems since the target algorithms are memory bounded.

These promising results lead us to envision new methodologies for SLP extraction based on improving the original SLP compiler with additional superword recombination stages.

References

[1] A. Bik, M. Girkar, P. Grey, and X. Tian. Efficient exploitation of parallelism on pentium iii and pentium 4 processor-based systems. *Intel Technology Journal*, 2001.

[2] D. Chaver, C. Tenllado, L. Piñuel, M. Prieto, and F. Tirado. 2D wavelet transform enhancement on general-purpose microprocessors: Memory hierarchy and simd parallelism exploitation. In *Int. Conf. of High Performance Computing (HiPC)*, pages 9–21, 2002.

[3] S. Fuller. Motorola's AltiVec technology. Technical Report ALTIVECWP/D, MOTOROLA, 1998.

[4] Intel. Intel architecture optimization. reference manual. http://developer.intel.com.

[5] Intel. Intel architecture software developer's manual. http://developer.intel.com.

[6] Ken Kennedy and John R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[7] A. Krall and S. Lelait. Compilation techniques for multimedia processors. *Int. Journal on Parallel Programing*, 28(4), 2000.

[8] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proc. of the ACM SIGPLAN 2000 Conf. on Programming language design and implementation (PLDI'00)*, pages 145–156, New York, NY, USA, 2000. ACM Press.

[9] S. Larsen, E. Witchel, and S. Amarasinghe. Techniques for increasing and detecting memory alignment. Technical Report MIT-LCS-TM-621, MIT, USA, 2001.

[10] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* Addison-Wesley, Massachusetts, USA, 1991.

Synthetic Kernel. Speedups achieved by the original SLP compiler on the Pentium 4 platform. LU and SLP stand for the contribution of loop unrolling and the overall speedup respectively.

size	LU	SLP
128	0.843	1.116
256	0.847	1.116
512	0.879	1.1
1024	0.883	1.075
2048	0.893	1.097
4096	0.902	1.079
144	0.844	1.116
288	0.849	1.12
576	0.86	1.083
1152	0.881	1.071
2304	0.896	1.078
4608	0.902	1.083

Table 3
Synthetic Kernel. Speedups achieved by the SLP and PT-SLP compilers on an Pentium4 platform. Speedups are calculated over an optimized scalar version that already incorporates *unroll-and-jam*.

N	UJ	SLP + UJ	PT-SLP
128	2.154	0.63	3.079
256	2.036	0.655	2.734
512	1.756	0.718	1.826
1024	2.176	0.625	1.491
2048	2.155	0.628	1.474
4096	2.151	0.629	1.474
144	2.171	0.621	3.267
288	2.127	0.624	3.212
576	1.849	0.696	1.77
1152	2.158	0.631	1.492
2304	2.131	0.631	1.516
4608	2.16	0.63	1.501

Table 5
Intra-component transform. Speedups achieved by *unroll-and-jam* and our PT-SLP compiler on the Pentium 4 platform.

N	UJ	PT-SLP
128	1.176	2.277
256	1.175	2.318
512	0.839	1.294
1024	1.117	1.418
2048	1.135	1.435
4096	1.112	1.43
144	1.193	2.267
288	1.121	2.104
576	0.885	1.342
1152	1.126	1.46
2304	1.099	1.456
4608	1.113	1.472

Table 2
Synthetic Kernel. Speedups achieved by the original SLP compiler on the G4 platform. LU and SLP stand for the contribution of loop unrolling and the overall speedup respectively.

size	LU	SLP
128	1.56	1.598
256	1.593	1.66
512	1.385	1.452
1024	1.35	1.418
2048	1.35	1.418
4096	1.352	1.419
144	1.562	1.597
288	1.598	1.665
576	1.365	1.433
1152	1.351	1.419
2304	1.348	1.418
4608	1.351	1.419

Table 4
Synthetic Kernel. Speedups achieved by the SLP and PT-SLP compilers on an the G4 platform. Speedups are calculated over an optimized scalar version that already incorporates *unroll-and-jam*.

N	UJ	SLP + UJ	PT-SLP
128	3.071	1.008	4
256	3.132	1.014	4.078
512	2.333	0.979	1.461
1024	2.116	0.98	1.326
2048	1.932	0.979	1.262
4096	1.772	1.016	1.256
144	3.062	1.013	4.075
288	3.167	1.01	4.152
576	2.254	0.984	1.378
1152	2.09	0.993	1.322
2304	1.93	0.994	1.255
4608	1.716	1.059	1.293

Table 6
Intra-component transform. Speedups achieved by *unroll-and-jam* and our PT-SLP compiler on the G4 platform.

N	UJ	PT-SLP
128	1.079	4.314
256	1.114	4.477
512	1.013	2.34
1024	1	1.985
2048	0.998	1.782
4096	0.99	1.577
144	1.077	4.28
288	1.091	4.136
576	1.006	2.191
1152	0.998	1.951
2304	0.998	1.76
4608	0.99	1.578

Applications

High Volume Colour Image Processing with Massively Parallel Embedded Processors

Jan Jacobs[a], Winston Bond[b] Roel Pouls[a], Gerard J.M. Smit[c]

[a]Océ Technologies BV

[b]Aspex Semiconductor Ltd

[c]University Twente

Currently Océ uses FPGA technology for implementing colour image processing for their high volume colour printers. Although FPGA technology provides enough performance it, however, has a rather tedious development process. This paper describes the research conducted on an alternative implementation technology: software defined massively parallel processing. It is shown that this technology not only leads to a reduction in development time but also adds flexibility to the design.

1. Introduction

Océ Technologies B.V. develops products and services for the professional printing market, from small format office printers (up to A3) to wide format design department printers (up to A0+).

Océ is investigating ways to reduce the rather long development trajectory for the printer's colour image processing subsystem, which is currently implemented with FPGA technology. The FPGA technology, consumes more development time than a software defined system, such as DSPs, general purpose processors and associative processors. This research is inspired by the potential advantages of a massively parallel software defined system, namely flexibility and shorter design cycles, while attaining equivalent or better performance (by scalable design).

Many of the image processing tasks in a printer show simple massively parallel processing. Therefore, Océ has performed research into applicability of massively parallel embedded processors with similar characteristics. This research has been conducted in co-operation with Aspex, a fabless semiconductor company specialising in high performance, software programmable, parallel processors based on associative technology [5].

The problem addressed in this paper is: "can we use associative processing for high performance colour image processing?"

In chapter 2 the reader is introduced to some important concepts like: colour image processing, associative processing and FPGAs. The next chapter deals with the specification of the problem, which is followed by the mapping onto the associative technology. Finally the results, among which a comparison with an FPGA implementation, are given.

2. Related Work

Colour Image Processing

Colour image processing deals with the transformation of an input colour image into an output colour image with suitable properties (e.g. for the printing process). Three important issues that will be discussed are: image representation, image transformation and desired properties with respect to printing.

- Images are represented as matrices in which each element contains a single integer for monochrome

or multiple integers for colour images. The resolution of an image, expressed in for example pixels/inch, directly relates to the dimensions of this matrix.

- Image transformations or operations, which change a pixel's value, may be divided into two categories. In the first category (point operation), for example, the result of thresholding a grey level pixel only depends on the value of the pixel itself. The second category is called neighbourhood operation. In this category pixels are taken from a certain environment in order to compute the resulting value of the pixel, e.g. edge detection.

- Printing high quality colour documents involves a carefully selected set of image transformations which are combined in a colour image pipeline. Examples are given in section 3.

Associative Processing

Traditional computers, rely upon a memory that stores and retrieves data by its address rather than by its content. In such an organisation (von Neumann architecture), every accessed data word must travel individually between the processing unit and the memory. The simplicity of this retrieval-by-address approach has ensured its success, but has also produced some inherent disadvantages. One is the von Neumann bottleneck, where the memory-access path becomes the limiting factor for system performance. A related disadvantage is the inability to linearly increase the performance of a unit transfer between the memory and the processor as the size of the memory scales up [1]. Associative memory, in contrast, provides a naturally parallel and scalable form of data retrieval for both structured data (e.g. sets, arrays, tables, trees and graphs) and unstructured data (raw text and digitized signals). An associative memory can be easily extended to process the retrieved data in place, thus becoming an associative processor. This extension is merely the capability of writing a value in parallel into selected cells [6]. Applications range from handheld gaming, multimedia, base stations, on-line transaction processing, image processing, pattern recognition and data mining [5].

Aspex's Linedancer is an implementation of a parallel associative processor. The approach taken by Aspex Semiconductor is to use many simple associative processors in a SIMD arrangement. Each of the 4096 processing elements on the Linedancer device has about 200 bits of memory (of which 64 are full associative) and a single bit ALU, which can perform a 1 bit operation in 1 clock cycle. Operations on larger data types take multiple clock cycles.

The aggregate processing power of Linedancer depends entirely on parallel processing. A 32-bit add will take many times the number of clock cycles taken by a high-end scalar processor, but due to the parallelism 4096 additions can be performed in parallel. Multiple Linedancer devices can be easily connected together to create an even wider SIMD array.

The Linedancer device (shown in Figure 1) includes an intelligent DMA controller, to ensure that data is moved in and out of the ASProCore concurrently with data processing, and a RISC processor, to issue high level commands to the ASProCore and to setup the DMA controller. All parts of the device run at the same clock frequency, which can be up to 400 MHz. A Linedancer is programmed in an extended version of C, with additional syntax for controlling the ASProCore.

FPGA

Currently FPGAs are used for the colour image processing. FPGA stands for *Field Programmable Gate Array* and denotes an integrated circuit which is programmed in the field as opposed to an *Application Specific IC* (ASIC). ASICs are typical used in high volume quantities because of the high development efforts, costs and manpower involved. As we are not targetting the consumer market, ASICs are not considered in this paper.

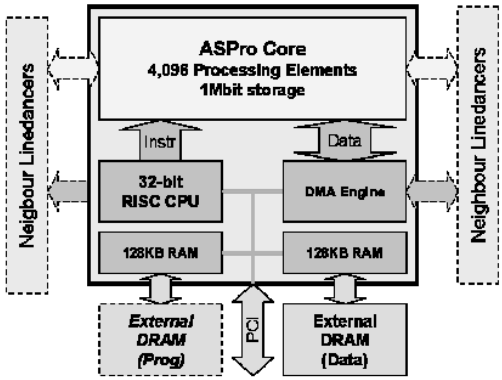


Figure 1. Aspex Semiconductor's Linedancer

From a computation viewpoint the FPGA offers a two dimensional array of configurable logic blocks which are capable of processing a two dimensional image. As the majority of image processing algorithms can be broken down into highly repetitive tasks, FPGAs present a very interesting alternative. An important property of an FPGA is that its throughput is better than the throughput of a von Neumann processor. This can be achieved because the individual logic cells of FPGAs map well to the individual mathematical steps involved in image processing.

FPGAs such as the Xilinx Virtex series [7] provide a large two-dimensional array of logic blocks where each block contains several flip-flops and look-up tables capable of implementing many logic functions. In addition, there are also dedicated resources for multiplication and memories that can be used to further improve the performance.

3. Specification of the Application

Functional process graph

For simplicity, it was decided to restrict the initial research to the most challenging components of the colour pipeline for a 30 pages per minute, 600 dots per inch, full colour printer. A block diagram of this simplified pipeline is shown in Figure 2. Also shown is the amount of communication between the blocks, indicated as bits per pixel.

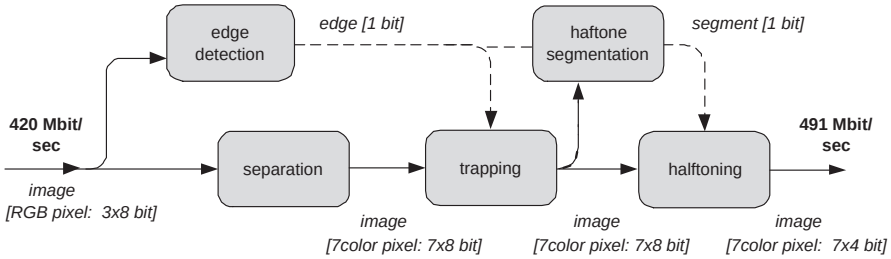


Figure 2. Simplified image processing pipeline

Separation

The Separation stage of the image processing pipeline, also known as colour space conversion, translates the RGB image data into the 7 toner colours that are available in the printer: black (K), blue (B), red (R), green (G), cyan (C), magenta (M) and yellow (Y).

Various algorithms exist for this task, but high quality colour space conversion is a highly non-linear operation [3]. The best results are obtained with a look-up table (LUT) based approach. The

look-up table is a large LUT of $2^{24=3\times8}$ entries of 7×8 bit = 940 Mbit in total.

Edge Detection

Edge detection is a neighbourhood operation which determines whether a pixel is an edge pixel or not. The Edge Detection module assists the other modules in making the right choices how to process a particular pixel [2]. The functionality is based on absolute differences in the RGB colour values between each pixel and its immediate neighbours (in a 3×3 kernel). In general such a neighbourhood operation can be specified by the absolute value of a convolution edge $=$ |kernel $\otimes$ image|[1]. For a small and symmetrical 3×3 kernel this convolution may be described by

$$\forall_{i,j\in N\times M}\text{Pixel}\,(i,j) = \sum_{k=-1}^{1}\sum_{l=-1}^{1}\text{kernel}(k,l)\cdot\text{Pixel}\,(i+k,j+l)$$

Figure 3 shows an example of a 3×3 kernel.

-1	-1	-1
-1	8	-1
-1	-1	-1

Figure 3. Sample edge detection kernel

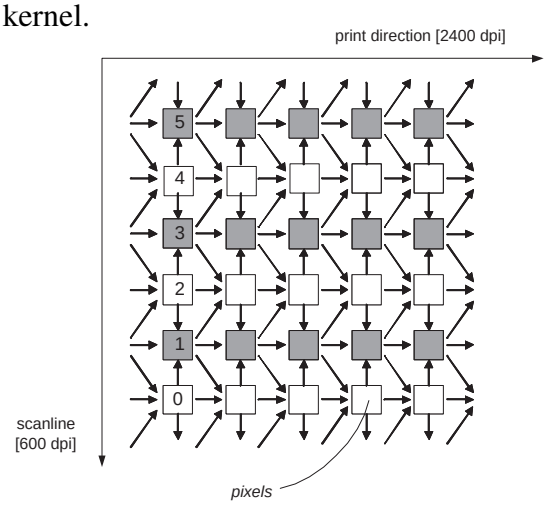


Figure 4. Half-toning error propagation, scanlines arranged horizontally

Trapping

The purpose of trapping is to enhance the print quality by reducing the visibility of small misalignments between the different mechanical components used to print each colour [4].

Trapping decreases the visibility of misalignments by creating an overlap between areas of the different toner colours.

One consequence of the trapping stage is that the imaging pipeline cannot process the 7 colour planes independently. Changes in one colour plane can cause changes in other colour planes.

Half-tone Segmentation

Half-tone segmentation controls how each colour channel has to be half-toned in order to select the best technique for half-toning further down the image pipeline. Half-tone segmentation is another edge detection operation, with many similarities to the Edge Detection stage. The main difference is that it must operate on the 7 colour data output by the trapping stage.

Half-toning

The purpose of half-toning is to render continuous tone information for a print engine, which has a lower tonal resolution than the input bitmaps. 8-bit image data can have 256 different values, but ink is binary – it is either printed or not [3].

[1] $\otimes$ stands for the convolution operator.

Printers overcome this problem by printing at a higher spatial resolution than the input bitmap. In our case, for example, the printer will print in one dimension 4 ink dots (sub-pixels) for each input pixel. A 600×600 dpi image will be printed using 600×2400 dots of each of the 7 colours, per square inch.

Half-toning has to translate 8-bit colour values into 4 ink dots per pixel and does this depending on the pixel being an edge or not. Pixels in a smoothly varying neighbourhood are treated by dithering, a technique which optimises grey level quality at the expense of some spatial resolution. Edges, however, are treated specially in order to retain the sharpness or spatial resolution: they spread the error between the desired colour and the realised colour around the pixels in the very close neighbourhood. See Figure 4 for this required spread of errors: the scanlines are arranged vertically and aggregate errors have to be propagated in horizontal direction.

Performance requirements

The processing of each pixel on the printed page is relatively simple, being mainly based on 3×3 convolution kernels. However, the total image processing pipeline is a challenging task because of the volume of data involved.

All images in the pipeline represent a 7K$\times$5K A4 bitmap page. Every stage must be performed on 35 million pixels and multiple colours. In the two edge detection stages alone there are 350 million (edge detection) operations for every printed A4 page and trapping adds even more.

Many of the tasks in the imaging pipeline can be implemented for many pixels in parallel. The next chapter describes how the image processing pipeline was implemented on the Linedancer parallel processing device.

Table 1 contains the processing requirements for a sequential implementation.

module	number of sequential operations / pixel
separation	3
edge detection	68
trapping	255
half-tone segmentation	30
half-toning	1162

Table 1
Processing requirements

4. Design

Because the size of the whole bitmap is much larger than the available storage space in the Linedancer we have to use a kind of bitmap tiling or patching. For a 4K processor array in which a single Processor Element (PE) deals with a single pixel, a 64×64 tiling scheme is used. For simple point operations each pixel has sufficient data to compute the result. However, for neighbourhood operations as in case of edge detection, the directly involved 8 neighbouring pixels have to be copied to each PE. This can be done for all PEs in parallel. A consequence of this approach is that pixels at the tile's boundary do miss pixel values so can not determine its value. For trapping, for example, only the 62×62 inner pixels can be computed effectively per tile. However, due to the subsequent neighbourhood like computation of half-tone segmentation an extra overlap band is needed, which yields an effective payload of 60×60 inner pixels.

For edge detection, trapping and half-tone segmentation we use the same square patching structure. However, the structure of half-toning breaks up the pipeline and forces us to use a second pass where all pixels are revisited. The intermediate results after half-tone segmentation are dumped into memory and are reloaded again but now using very slim, 1 pixel wide, scanline patches. The reason for this is that the error inputs (see Figure 4) must be added to each pixel in a line before it can be half-toned and its error output can be calculated. Every pixel propagates an error output to 3 neighbours. First the even pixels send an error component to the odd pixels in the current line and the even pixels on the next line. Then the accumulated error of the odd pixels in the current line send errors to the 3 neighbour pixels on the next line.

As mentioned we must process the error propagation one scan line at a time. First doing the calculations for the even pixels, then the odd pixels, while working from left to right.

We can fit an A4 page height (7K pixels) into one Linedancer by packing an odd/even pair of pixels into each processing element; 12% of the PEs remain unused.

This tiling strategy described above, implies a two pass process. All modules except for half-toning are processed in a 64×64 square pixel tile or patch with overlap to accommodate convolution operations like in edge detection. The initial time budget for processing a single patch can be derived by dividing the total number of clocks in the 2 sec/page (2×400 MHz) by the number of patches needed (35 Mpixels / 60×60)[2] yielding a budget of 82K clocks per patch for a single Linedancer.

All modules except for half-toning are run subsequently with a square patch and the intermediate result of pass 1 is dumped to the RAM, see Figure 5.

Then the second pass is started in which complete scanlines are processed in parallel.

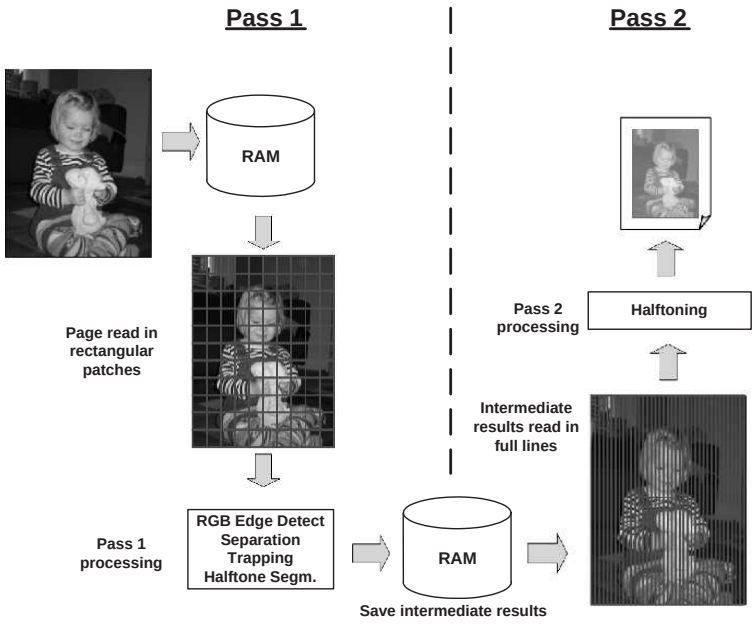


Figure 5. Overview of the 2 pass pipeline

[2]Effective area of a patch is $(64 - 4) \times (64 - 4)$ due to the 2×1 pixel overlap at each side.

5. Results and Comparison with FPGAs

In this section the results of the previously elaborated modules are combined in order to formulate a conclusion on the feasibility of functionality and timing of the Linedancer implementation. It serves furthermore as a basis for comparison with FPGA technology.

Pass 1 consisting of loading RGB, edge detection, separation, trapping and half-tone segmentation takes 16K5 cycles per patch. From these modules Separation runs on the DMA controller in parallel with the other modules. Half-toning in pass 2 takes up 2K5 clocks per line per colour (see Table 2).

ASProCore		DMA			
module	number of cycles / patch	module	number of cycles / patch	number of patches / page	number of cycles / page
PASS I					
edge detection trapping half-tone segmentation	2K2 7K3 5K	load tile separation	2K 6K5		
	16K5 cycles per patch			9K7	161M
PASS II					
half-tone	17K5 cycles per line			5K	87M
TOTAL					248M

Table 2
Performance estimate

The overall processing time is 248M cycles per A4 page, which is equivalent to 0.62 seconds per page, with a single Linedancer at 400 MHz. This is well below the required 2 seconds per page.

Illustrative for the power of massively parallel computing is the speedup compared to the sequential implementation as shown in Table 1. Although the processing capacity of each PE is much lower than a von Neumann processor, the number of processors working in parallel yield large speedups. Parallelism accounts for the speedup although the clock-speed of the Linedancer is lower, see Table 3.

module	number of seq cycles[3]/ pixel	number of cycles / pixel	speedup
separation	3	6K5 / 3600 = 1.81	1.66
edge detection	68	2K2 / 3600 = 0.61	111
trapping	255	7K3 / 3600 = 2.03	126
half-tone segmentation	30	5K / 3600 = 1.39	21.6
half-tone	1162	17K5 / 7K = 2.5	465

Table 3
Measured speedup

The productivity benefits of using an associative processing approach to this problem rather than using FPGA technology are shown in Table 4. The ratio of development effort for an FPGA versus

[3]Based on single cycle operations

Linedancer is estimated at 2:1 (including coding, testing, PCB design etc.), assuming a person with domain and target hardware experience.

technology	development effort [man days]	execution speed [ppm]
2 way SMP Intel Pentium Xeon[4]	10-20	2-30
FPGA Spartan2E[5]	100	30
Linedancer	50	90

Table 4
Comparison

6. Conclusions

An associative processor combines the speed of FPGAs with high-level software programmability and flexibility.

A single Linedancer device is capable of implementing a colour image processing pipeline at a rate of 90 pages per minute, well above the required 30 pages per minute [ppm]. Large speedups can be realised compared to the sequential case: the speedup in operations/sec can go up as far as 400 (up to 80 in execution time).

A key issue in the design is how to partition the 35M pixels of a page into 4K chunks for processing. This apparently simple problem is complicated by the conflicting requirements of the various 3×3 kernel operations and the error propagation in half-toning.

Software defined systems enable fast developments. The development of code for a PC based solution is faster. But when real time performance is critical, and the choice is between FPGA or Linedancer, than the use of the latter reduces the design cycle by a factor of 2.

Due to the inherent scalable architecture the performance can scale with the number of processors without changing the code (e.g. delivering more productivity, more resolution, more colours).

References

[1] Deszo Sima, Terence Fountain, Peter Karsuk: Advanced Computer Architectures: A Design Space Approach. Addison Wesley. ISBN 0201422913. 1997.
[2] Gonzales, Woods: Digital Image Processing (2nd Edition). Prentice Hall. ISBN 0201180758. 2002.
[3] Kang: Color Technology for Electronic Imaging Devices. SPIE-International Society for Optical Engine. ISBN 0819421081. 1997.
[4] Alex Vakulenko: `http://www.oberonplace.com/draw/trapping/`. 1999.
[5] Aspex Semiconductor Ltd: `http://www.aspex-semi.com/products/downloads/aspex-_technology_background.pdf`. 2004.
[6] Anargyros Krikelis, Charles C. Weems: Associative processing and processors. IEEE Computer, Volume 27 , Issue 11 (November 1994), Pages: 12 – 17.
[7] Xilinx: `http://www.xilinx.com/products/virtex4/overview.htm`.

[4]Numbers represent normal as well as the optimised case
[5]The system could be build using $\pm$ 5 Spartan XC2S400E devices

Parallel Endmember Extraction Techniques Applied to a Self-Organizing Neural Network for Hyperspectral Image Classification

D. Valencia*[a] A. Plaza[a], R.M. Pérez[a], M.C. Cantero[a], P. Martínez[a], J. Plaza[a]

[a]Neural Networks and Signal Processing Group (GRNPS), Computer Science Department,
Computer Architecture and Technology Section Avda. de la Universidad s/n,
University of Extremadura, E-10071 Cáceres, Spain

1. abstract

Advances in remote sensing technology have recently led to the development of hyperspectral sensor instruments, capable of collecting hundreds of images for the same area on the surface of the Earth at different wavelengths in the electromagnetic spectrum. Such images, which can be considered spatially and spectrally continuous, are characterized by their extremely large dimensionality. The identification of pure spectral constituents (called *endmembers*) in those images is considered to be a crucial task in hyperspectral data exploitation. Endmember extraction algorithms are computationally very expensive, due to the fact that they are based on complex mathematical operations. However, both the intrinsic properties of the image data and regularities in computations make these algorithms suitable for parallel implementation. In order to exploit the proposed techniques in applications that require a response in near real time, this paper investigates parallel implementations of a combined morphological/neural classification algorithm for hyperspectral imagery. The proposed implementation has been developed by considering two possible data-domain partitioning schemes: spatial-domain parallelism and spectral-domain parallelism. The performance of the parallel algorithm is tested on Thunderhead, a 256-processor massively parallel Beowulf cluster at NASAs Goddard Space Flight Center in Maryland.

2. Introduction

Last generation hyperspectral sensors have recently demonstrated their potential in land cover identification and characterization applications [1]. Each pixel collected by those sensors is given by a high-dimensional vector of values that provides a "spectral signature", which can be used to accurately characterize the composition of each site. A common technique for hyperspectral image classification relies on the identification of pure spectral signatures, often called spectral "endmembers" due to the fact that they are typically located on the corners of the hyperdimensional cube defined by the data volume. These pure signatures can be used to produce both *hard* and *soft* classifications, where a pixel vector may be classified into a single pure class or several mixed classes with different sub-pixel proportions.

One of the most successful approaches to endmember extraction in the literature has been the Automated Morphological Endmember Extraction (AMEE) algorithm [2], which is fully automated and does not require any data pre-processing. This technique successfully integrates both the spatial and spectral information in the data to conduct a multidimensional endmember search. The algorithm is computationally expensive due to complex mathematical operations involved. However, both the intrinsic properties of the image data and the regularities in endmember-based computations make

*Corresponding author. Contact E-mail: davaleco@unex.es

this algorithm highly amenable to parallel implementation. It should also be noted that the endmember pixels provided by endmember extraction algorithms such as AMEE are suitable to be used as input information for other applications. For instance, there are many situations where a detailed knowledge of image endmembers is not sufficient to extract a detailed land-cover classification map. In this context, artificial neural networks (ANNs) have demonstrated to be a powerful tool for land-cover classification because the information provided by ANNs can not only be used to provide *hard* labels, but also to obtain *soft* classification labels, e.g., by taking into account the degree of membership or similarity of a certain input pattern (pixel vector) to a certain output class (endmember). In the field of ANN-based remotely sensed data interpretation, Kohonen's self-organizing map (SOM) has been widely recognized as a very powerful tool to perform both hard and soft classification. This model is based on an unsupervised learning strategy that does not require any previous test samples [3,4]. Again, one of the main restrictions of SOM-based analysis is its high computational complexity, which is a serious drawback in applications that require a response in near real-time, such as those aimed at detecting and/or tracking natural disasters such as forest fires, oil spills, and other types of chemical contamination.

In this paper, we develop a parallel implementation of a combined AMEE/SOM (morphological/neural) approach to hyperspectral image classification. Although a few parallel algorithms for hyperspectral imaging exist in the literature [5], our parallel approach is the first one that integrates both spatial and spectral information in simultaneous fashion. It relies on data-parallel domain decomposition techniques aimed at minimizing inter-processor communication and maximizing load balance. It should be noted that the proposed method relies on well-known strategies, which have been widely used in the past for handling parallel computations in other research areas including High-Performance Fortran (HPF) and parallel skeletons [6,7]. However, the application of the proposed data-parallel strategy to hyperspectral imaging is new and represents a novel contribution. The paper is structured as follows. Section 3 describes the algorithm proposed for parallelization. Section 4 discusses key features related to the parallelization of the algorithms and their implementation. Section 5 reports parallel performance and classification results achieved by our combined AMEE/SOM approach. Finally, section 6 concludes with some remarks and hints at plausible future research.

3. Algorithms

In this section, we briefly address the fundamental properties of the individual AMEE and SOM techniques. These methods are available in the open literature and we will not expand on their detailed properties here, but relevant hints for their parallelization will be pointed out.

3.1. Automated morphological endmember extraction (AMEE)

In order to define extended morphological operations in hyperspectral imaging, we first impose an ordering relation in terms of spectral purity in a set of neighboring pixel vectors lying within a kernel neighborhood, known as structuring element (SE) in mathematical morphology terminology [2]. Using the two basic morphological operations illustrated in Fig. 1, the AMEE algorithm calculates a *morphological eccentricity index*(MEI) by comparing the output of the dilation to the output of the erosion for each pixel in the input data, using the SE in sliding-window fashion [2]. The MEI index is updated by repeating the above procedure by several algorithm iterations, where the result of the morphological dilation replaces the input data at the end of each iteration. The complexity of AMEE algorithm is $O(p_f x p_B x I_{MAX} x N)$, where p_f is the number of pixels of the input hyperspectral image f; p_B is the number of pixels in the structuring element; I is the number of iterations executed by

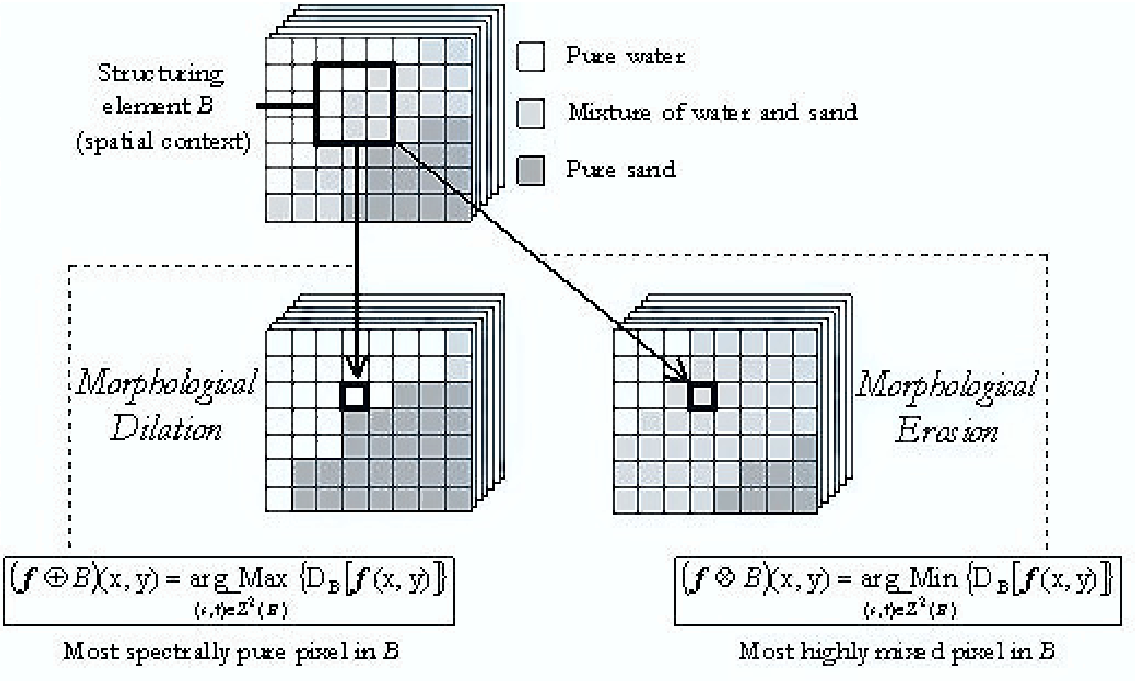


Figure 1. Morphological operations extended to hyperspectral imagery.

the algorithm; and N is the number of spectral bands. This results in very high computational complexity, in particular, when the value of N is very large [2]. Parallelization of AMEE must take into account the data dependencies introduced by the adopted sliding window-based approach, illustrated in Fig. 1.

3.2. Self-organizing map (SOM)

The neural model proposed in this work consists of N input neurons and M output neurons [3], where N is the dimensionality of the input pixel vectors, and M is the number of endmembers or class prototypes provided by AMEE algorithm. The network consists of two layers, with feedforward connections from the input layer to the output layer and a set of associated connection weights, arranged in a matrix that will be denoted hereinafter as W_{MxN}. The working procedure of the network is given by two different stages: clustering and training. In the former, the endmembers found by AMEE are presented to the network so that feedforward connections change and adapt to the information provided by the spectral data. In the training stage, feedforward connections project input patterns onto the feature space, and the Euclidean distance is used to identify a winning neuron. This procedure is summarized below:

1. Weight initialization. Normalized random values are used to initialize the weigth vectors: $w_i^{(0)}$, with $i = 1, 2, ..., M$.

2. Training. In this work, this step is accomplished by using AMEE-generated endmember signatures.

3. Clustering. For each input pattern x, a winning neuron i^* is obtained at time t by using an Euclidean distance-based similarity criterion, i.e., $i^*[x] = \min_{1<=j<=M} ||x - w_j||^2$.

4. Weight adjustment. The winning neuron (and those neurons in the neighborhood of the winning

one) adapt their weights using the following expression, where $\alpha(t)$ and $\sigma(t)$ are the learning and neighbouring functions, respectively. $w_i^{(t+1)} = w_i^t + \sum_{t'=t_0}^{t_{max}} \alpha(t')\sigma(t')(x - w_i^{(t)})$

5. Stopping rule. The SOM algorithm terminates as soon as a pre-determined number of iterations, t_{max}, has been accomplished.

4. Parallelization

Two types of data parallelism can be exploited in the proposed algorithm: spatial-domain parallelism and spectral-domain parallelism. Spatial-domain parallelism subdivides the input image into multiple blocks made up of entire pixel vectors, and assigns one or more blocks to each processing element (PE). On other hand, the spectral-domain parallel paradigm subdivides the whole multiband data into blocks made up of contiguous spectral bands (sub-volumes), and assigns one or more sub-volumes to each PE. In the following, we provide a discussion on the two types of parallelism above and their impact on the individual steps (morphological/neural) of the proposed method.

4.1. Parallelization of the morphological algorithm

In previous work, we have analyzed several parallelization strategies for the morphological stage of our combined algorithm [8,9]. Those studies revealed that, when the input data partitioning is accomplished in the spectral domain, the local spatial information of the images remains together. This is an important property for parallelization of image processing algorithms of the window-moving type, because this paradigm allows division of the input data into sub-volumes that can be processed independently by the AMEE algorithm. Also, this approach does not need to hold any replicated information to complete the calculations involved in the morphological process. Despite the above remarks, our previous work [9] has demonstrated that the spectral domain decomposition paradigm is not suitable (in general) for hyperspectral imaging applications. The main reason is that most hyperspectral imaging techniques consider the spectral information contained in each pixel vector as a unique entity. In other words, a spectral domain decomposition paradigm would break the spectral identity of the data because each pixel vector would be split amongst several PEs. If we take into account the fundamental characteristics of the morphological algorithm, the selection of a partitioning scheme in the spatial domain is critical for the success of our parallelization.

Several reasons justify our decision to implement a spatial domain-based partitioning framework. First, this strategy retains the spatial/spectral information, a desired property in a combined approach such as the one implemented by the proposed morphological algorithm. Since the resulting partitions are composed of spatially adjacent pixel vectors, the application of a sliding window-based approach can be accomplished in parallel with minimum changes to the original algorithm, thus enhancing code reusability and portability. A second reason has to do with the cost of inter-processor communication. In spectral-domain parallelism, the SE-based calculations made for each hyperspectral pixel need to originate from several PEs, and thus require intensive communications. A final major reason is that spatial information is particularly relevant in the local neighborhood around each pixel. Subsequently, partitioning in the spatial domain guarantees that spatial/spectral information can be retained at no extra cost by the proposed parallelization strategy [8]. To conclude this subsection, we emphasize that the main drawback of the proposed parallelization strategy for the morphological algorithm is the need to replicate information in order to reduce inter-processor communication [9]. However, we have experimentally proved that the cost of processing redundant information is insignificant compared to the cost of transmitting the boundary data. The introduction of replicated data introduces border-handling and overlapping issues which are simply resolved by

considering those pixels inside the local processor domain only for the MEI calculation [8].

4.2. Parallelization of the neural algorithm

In order to parallelize the SOM algorithm, we face similar problems than those already raised in the previous subsection. A straightforward approach to parallelization of the neural algorithm is to simply replicate the whole neural network architecture, which is a feasible approach due to the random nature of the initial weights of the network. However, this option would result in the need for very complex rules of reduction, and also in integrity hazards [4]. Taking into account our previous studies [4,8] and considering the relatively small size of the training set, we have experimentally tested that the overhead usually takes place in the training stage (i.e, in the form of Euclidean distance calculations and adjustment of weight factors). This fact makes partitioning of the weight matrix W_{MxN} a very appealing solution to reduce the computation time. Again, two main alternatives can be adopted to carry out such partitioning: (1) Division by input neurons (endmembers/training patterns); or (2) Division by output neurons (class prototypes). It should be noted that, in the latter case, the parallelization strategy is very simple. Quite opposite, when the former approach is adopted, there is a need to communicate both the calculations and the intermediate results among different processors. This introduces an overhead in communications that may significantly slow down the algorithm; according to our preliminary experiments, this option could even give worst results than those found by the sequential version of the SOM algorithm. On the other hand, the partitioning scheme based on dividing by class prototypes only introduces a minor communication overhead. This approach creates the need to introduce a broadcast/all-reduce protocol in order to obtain the class prototype through local minimum calculations, in batch-mode processing fashion. The winner neuron for each pattern needs to be tallied, and subsequent modifications for the weight update factor also need to be stored for further addition/subtraction. This approach also allows us to directly obtain the winner neuron at each iteration without the need for any further calculations. It also facilitates a pleasingly parallel solution which takes full advantage of the processing power available in the considered parallel architecture while, at the same time, minimizing the overhead introduced by inter-processor communications.

At this point, we must emphasize that the proposed parallel scheme still creates the need to replicate calculations to further reduce communications. However, the amount of replicated data is limited to the complete training pattern set, which is stored at every local processor along with administrative information, such as the processor that holds the winner neuron, the processor that holds neurons in the neighborhood of the winner neuron, etc. Such information can also be used to reduce the communication overhead even further. For instance, we have considered two different implementations of the neighborhood modification function $\sigma(t')$: the first one is applied when a node is *inside* the neighborhood of the winner neuron, while the second is used when the node is *outside* the domain of that processor. To assess integrity of the considered function, a look-up table is locally created at each processor to tally the value of $\sigma(t)$ for every neuron pair. While in the present work the neighborhood function is gaussian, i.e., $\sigma(t) = e^{-\frac{\left|i^* - i\right|}{t}}$, other functions may also be considered as well [4]. In any regard, it is important to emphasize that when the neighborhood function is applied to the processor that holds the winner neuron, the neighborhood function is used in its traditional form. Quite opposite, when the function is applied to other processors, a modified version is implemented to average the distances to all possible winner neurons. This approach reduces the amount of communications and represents a more meaningful and robust neighborhood function [4]. As a final major remark, our MPI-based implementation makes use of blocking primitives to ensure that all processors are synchronized. This prevents integrity problems in the calculations related with

W_{MxN}.

5. Experimental results

The proposed parallel algorithm has been implemented in the C++ programming language using calls to message passing interface (MPI), where the MPICH 1.2.6 version was used in experiments due the demonstrated flexibility of this version to migrate the code to different platforms. The parallel algorithm has been tested on Thunderhead, a massively parallel Beowulf cluster at NASAs Goddard Space Flight Center in Maryland, where our parallel code is currently being exploited in various Earth-based remote sensing applications. Thunderhead is composed of 256 dual 2.4 Ghz Intel Xeon nodes, each with 1 Gb of memory and 80 Gb of main memory. The total peak performance of the system is 2457.6 Gflops. Before discussing the parallel performance achieved by the proposed algorithm, we briefly describe a hyperspectral scene (designated by AVIP92) that will be used for validation purposes in this work. The scene was collected by the NASA/JPL AVIRIS system [1] over a small area (145 lines by 145 samples) over the Indian Pines agricultural test site in Northwestern Indiana (available online from http://dynamo.ecn.purdue.edu along with 16 mutually exclusive ground-truth classes). Although the scene represents a challenging classification problem, the proposed algorithm achieved 90% overall accuracy and high individual test accuracies when applied to this scene. Fig. 2 shows the parallel performance of the morphological and neural algorithms, displayed separately for clarity. The two considered performance measures are the speedup and the parallel efficiency. In order to compute the speedup, we approximate the time required to complete a task on N parallel processors using $T(N) = A_N + \frac{B_N}{K}$ where A_N is the sequential (non-parallelizable) portion of the computation and B_N is the parallel portion. In the morphological algorithm, A_N is given by the sequence of operations implemented by the partitioning module, and B_N refers to the endmember extraction procedure. In the neural algorithm, A_N corresponds to the generation of random weight values, while B_N is dominated by the training procedure. We can define the speedup for N processors as $S(N) = \frac{T(1)}{T(N)} \approx \frac{A_N+B_N}{A_N+(\frac{B_N}{N})}$ where $T(1)$ denotes single processor time. Using the definitions above, we can further compute the parallel processing efficiency, defined as the actual speedup divided by the number of processors, i.e., $E_N = \frac{S_N}{N}$. As shown by Fig. 2, the

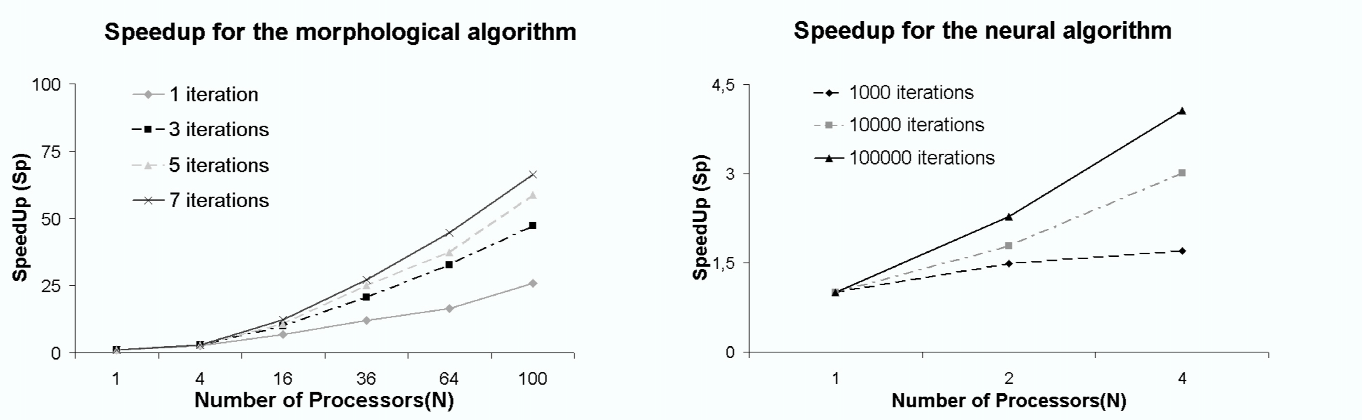


Figure 2. Speedup/parallel efficiency achieved by morphological/neural algorithms on Thunderhead.

morphological algorithm scales reasonably well on Thunderhead. This is because it takes advantage of some of the intrinsic characteristics of window-moving image processing algorithms, such as spatial and temporal data locality that result in cache reuse. The best speedup compromise for $I = 7$ algorithm iterations was achieved for $N = 16$ processors, with $S_{16} = 12.33$ and $E_{16} = 0.77$. The degradation in parallel efficiency as the number of processors is increased is likely due to the effect of redundant computations. On the other hand, Fig. 2 also reveals that, although parallelization of the neural algorithm is more complicated *a priori* due the expected impact of communications, the parallel neural code also scales relatively well (for a reduced number of processors). It should be noted that the cost of communications in the parallel neural algorithm cannot be reduced by introducing redundant computations, as it was the case in the morphological algorithm. Even though the amount of data to be exchanged is minimized by the proposed parallel neural strategy, we still had to deal with the size of the minimum transfer unit (MTU) of the communication network, a parameter that is not easily adjustable in the Thunderhead system. In future developments, we are planning on incorporating techniques able to automatically adjust the size of the MTU according to the properties of the input data. For instance, the domain of a single batch-mode iteration could be expanded to several network epochs (with all training patterns involved at each one) instead of just one epoch as in the current implementation. This could lead to much better data compaction inside the considered MTU. Also, we have detected that the parallel efficiency achieved for large training sets is significantly higher than that found for smaller training sets. This is because computations clearly dominate communications in this case, thus greatly enhancing the granularity of the parallel computation. As one would expect, the use of large training sets also results in much higher classification accuracies by the SOM neural network.

Although only results with 4 processors are reported in this work, we also observed that increasing the number of processors introduced fluctuations in the achieved speedups with significant drops in parallel efficiency. This is due in part to the scheduling policies implemented in the Thunderhead cluster, which tend to assign high priority to jobs that require a very large number of processors. Even in spite of the above limitations, our measured speedups reveal slight superlinear scaling effects in some cases, probably due to cache reuse (e.g., when 10^5 training iterations were considered, values of $E_2 = 1.135$ and $E_4 = 1.01$ were measured). This reveals that cache spatial and temporal locality could be partially used to overcome the limitations imposed by excessive communications. The above results also lead us to believe that the best configuration for the parallel SOM algorithm is likely to be achieved when most neural network partitions fit completely in the local processor caches. Further experimentation, however, is highly desirable in order to adapt the parallel properties of the neural algorithm to those observed in the morphological algorithm. In particular, there is a need to balance the combined computing power achieved by the pool of processors employed by the morphological algorithm and those used by the neural algorithm in the same algorithm run. This feature brings out new exciting future perspectives, such as the possibility to launch multiple neural-based classifiers in parallel. Such multiple classifier-based processing framework represents a completely novel data analysis paradigm in hyperspectral imaging, and previously looked too computationally complex to be developed in practical applications.

6. Conclusions

The aim of this paper has been the parallel implementation on high performance computers of an innovative morphological/neural technique for unsupervised classification of hyperspectral data sets. We show that parallel computing at the massively parallel level, supported by message passing, provides a unique framework to accomplish the above goal. For this purpose, computing systems

made up of arrays of commercial off-the-shelf computing hardware are a cost-effective way of exploiting this sort of parallelism in remote sensing applications. The two discussed parallelization strategies (morphological/neural) provide several intriguing findings and parallel design considerations that may help hyperspectral image analysts in selection of efficient algorithms for specific applications. Further, the proposed parallel strategies offer an unprecedented opportunity to explore methodologies in fields that previously looked to be too computationally intensive in practice, due to the inmense files common to remote sensing problems. The combination of this readily available computational power and the new perspectives introduced last generation sensor instruments may introduce major changes in the systems currently used by NASA and other agencies to process Earth and planetary remotely sensed data.

References

[1] R.O. Green et al.:"Imaging spectroscopy and the airborne visible/infrared imaging spectrometer (AVIRIS)". *Remote Sensing of Environment*, vol. 65, pp. 227-248. 1998.

[2] A. Plaza, P. Martinez, R. Perez and J. Plaza.: "Spatial/spectral endmember extraction by multidimensional morphological operations". *IEEE Transactions on Geoscience and Remote Sensing*, vol. 40, no. 9, pp. 2025-2041, Sept. 2002.

[3] T. Kohonen: "Self-Organizing Maps". *Springer*, Berlin, Heidelberg, 1995. (Second Edition 1997).

[4] P. Martinez, P.L. Aguilar, R. Perez and A. Plaza.: "Systolic SOM Neural Network for Hyperspectral Image Classification" in: *Neural Networks and Systolic Array Design*. Edited by D. Zhang and S.K. Pal. World Scientific, pp. 26-43, 2002.

[5] T. Achalakul and S. Taylor: "A distributed spectral-screening PCT algorithm". *Journal of Parallel and Distributed Computing*, vol. 63, pp. 373-384, 2003.

[6] M. Chandy, I. Foster, K. Kennedy, C. Koelbel, and C-W. Tseng.: "Integrated Support for Task and Data Parallelism". *International Journal of Supercomputer Applications*, vol. 8, pp. 80-98, 1994.

[7] M. Cole. Skeletal Parallelism home page. Available online: http://homepages.inf.ed.ac.uk/mic/Skeletons/

[8] A. Plaza, D. Valencia, P. Martinez and J. Plaza.: "Parallel Implementation of Algorithms for Endmember Extraction from AVIRIS Hyperspectral Imagery". *XIII NASA/Jet Propulsion Laboratory Airborne Earth Science Workshop*. Pasadena, CA, USA, 2004.

[9] D. Valencia, A. Plaza, P. Martinez and J. Plaza.: "On the Use of Cluster Computing Architectures for Implementation of Hyperspectral Analysis Algorithms". *Proceedings of the 10th IEEE Symposium on Computers and Communications*, pp. 995-1000, Cartagena, Spain, 2005.

[10] J. Dorband, J. Palencia and U. Ranawake.: "Commodity computing clusters at Goddard Space Flight Center". *Journal of Space Communication*, vol. 1, no. 3, pp. 60-75, 2003.

JPEG2000 Optimization in General Purpose Microprocessors*

C. García[a], C. Tenllado[a], L. Piñuel[a], M. Prieto[a]

[a]Dpto. de Arquitectura de Computadores y Automática
Universidad Complutense, 28040 Madrid, Spain
e-mail: {garsanca, tenllado, lpinuel, mpmatias}@dacya.ucm.es

We have addressed in this paper the efficient implementation of the JPEG2000 in high-end microprocessor. From a computational perspective, the demands of this new standard, which is substantially more complex than its predecessor, makes this goal a challenging but extremely important task. Previous work about this topic has focused on data locality issues. Our efforts are aimed at improving the exploitation of *Multimedia ISA Extensions* and *Simultaneous Multithreading*. Performance results obtained on an Intel Pentium 4 processor show impressive speedups, that range from 1.9 to 22 depending on the target image size.

1. Introduction

Increasing focus on multimedia applications has prompted many researchers to design efficient image and video coding systems. An outstanding example of this effort is the JPEG2000, the latest series of standards from the JPEG committee. Apart from achieving higher compression rates than its predecessor (popularly referred to as JPEG), it is capable of handling many more aspects than simply making the digital image files as small as possible. However, from a computational perspective it is substantially more complex than JPEG, which makes its implementation a challenging task.

In the computer architecture area, this focus has also prompted the adaptation of general purpose architectures for multimedia workloads. Basically, functional units have been enhanced with *Subword Level Parallelism* (SLP) capabilities and the instruction-set architectures (ISA) have been extended to include new instructions (*Multimedia ISA Extensions*) that operate on packed data. Well-known examples are the Intel's SSE/SSE2/SSE3 ISA extensions of the IA-32 family [23] and the IBM-Motorola's Altivec [7]. However, the use of such extensions is difficult. Compiler technology is still not highly developed in this field and programmers are usually restricted to using in-line assembly, intrinsic functions or specialized libraries [18].

Aside from these extensions, multimedia codes can also take advantage of the additional levels of parallelism available in high-end microprocessors, such as *Thread-Level Parallelism* (TLP). *Single-Chip Multiprocessors* (CMP) are expected to soon be ubiquitous [3] and most superscalar-style cores likely will have some form of *Simultaneous multithreading* (SMT). In particular, SMT has already been incorporated into the Intel's Pentium 4 and Xeon families [14] and into the IBM's Power5 [11].

Our main goal in this paper is to study how to adapt the JPEG2000 explicitly to take advantage of both SLP and TLP parallelism on SMT architectures. The limitations of current compiler technology and the importance of this standard, makes this study of great practical interest. In addition, it also provides certain insights about the potential benefits of these relatively new capabilities and how to take advantage of them, which can help to develop more efficient compilers.

Our target code is a reference implementations of the standard, known as *JasPer* [1], which implements the codec specified in the JPEG2000 Part-1 standard (i.e., ISO/IEC 15444-1). For the

*This work has been supported by the Spanish government research contract TIC 2002-750 and the Hipeac European Network of Excellence

sake of conciseness, we have focused on the lossy compression process. Nevertheless, the proposed methodology can also be applied to both the lossless mode and the decoding process. Experiments have been performed on an Intel Pentium 4 running at 3.4 Ghz (2MB L2 cache, 1Gb DDR400).

The rest of the paper is organized as follows: some related work is summarized in Section 2; Sections 3 and 4 describe the proposed optimizations and present some performance results. Finally the paper ends with some conclusions.

2. Related Work

A significant amount of work on the optimization of the *lifting*-based 2D discrete wavelet transform (DWT) has been performed in recent years within the JPEG2000. This interest is generated by the considerable percentage of execution time involved in this component of the standard (around 40-60% according to some authors [19]). Most optimizations have been focused on improving cache reuse [15,16,4]. In [15], authors addressed the optimization of *JasPer* and *jj2000* (another reference implementation of the JPEG2000 written in java) by means of traditional *loop-tiling* and *array-padding* techniques (denoted by them as *aggregation* and *row-extensions*). Despite the relative simplicity of the proposed optimizations, the combined effect of both techniques on the DWT vertical filtering (the one that lacks spatial locality if the image is stored following a row-major layout) achieves a speed-up factor of around 10 for large image sizes. This tremendous improvement translates into an overall coding time reduction of around 2. This research was extended in [16] to shared-memory symmetric multiprocessors, applying the general principles of data-domain decomposition. Apart from *loop-tiling* techniques, [4] investigates the use of specific array layouts as an additional means of improving data locality.

Regarding SLP, we should mention [21], where a fixed-point implementation of the DWT has been vectorized using Intel's MMX. Related work centered on multithreaded architectures is also scarce. We can mention [12], in which a load adaptive approach for fine-grain multithreading architectures is proposed.

3. Exploiting SLP in the JPEG2000

Conventional vectorization techniques were designed during the 70's and the 80's to extract parallelism from computational intensive Fortran programs. Today, these techniques are being adapted to support the short-vector processing capabilities found in modern microprocessors and to take into account the requirements of multimedia workloads. In the commercial marketplace, we should highlight for example the continuous efforts being made by Intel, IBM and the Portland Group in their respective compiler infrastructures [2,9,8]. Within the academic community, we should mention the research of Larsen et al. [13] and the new release of the open-source compiler GCC 4.1, which has been significantly enhanced to support automatic vectorization [17].

Despite these extensive efforts, state-of-the art compilers cannot automatically vectorize any component of the *JasPer* library from the scratch.

3.1. Code Analysis

The compiler limitations mentioned above forced us to conduct an in-depth code analysis to find out which procedures are susceptible of SLP exploitation. The analysis was guided by the hints provided by the Intel C/C++ Compiler v.8.1 [10], which help us to identify vectorization inhibitors, and by an extensive profiling that helped us to select the most demanding components. Instead of using the original *JasPer*, our baseline code is a hand-tuned version, which already includes a

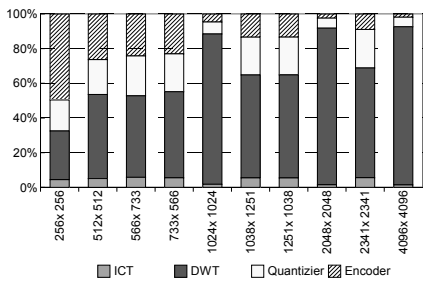


Figure 1. Execution time breakdown of the Original *JasPer's* lossy compression process for different images sizes.

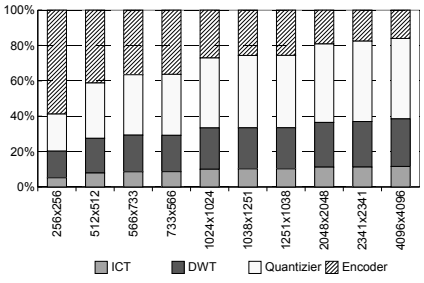


Figure 2. Execution time breakdown of our cache-aware implementation of the *JasPer's* lossy compression process for different images sizes.

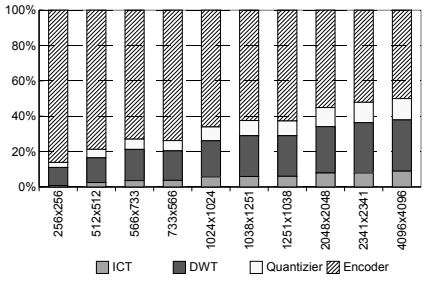


Figure 3. Execution time breakdown of our cache-aware implementation of *JasPer's* lossy compression process turning on SLP optimizations.

cache-aware DWT implementation (more details in Section 3.3).

As a result of this analysis, we identified three potential candidates in the lossy compression process: the *Irreversible Color Transform* (ICT), the *Discrete Wavelet Transform* (DWT) and the *Quantization* process.

The *encoding/decoding* stages have not been considered since they are characterized by substantial data and control dependencies, as well as irregular memory access patterns, limiting opportunities for fine-grain data parallelism exploitation.

Figures 1 and 2 show execution time breakdowns of the original and our baseline *JasPer's* implementation respectively. The reported percentages correspond to the coding of the *Lena* color image (treated as a single tile), using lossy compression with full bit-rate. The image sizes have been chosen following the results reported in [4]. Each bar shows the relative execution cost of the most important components of this process (excluding I/O operations).

It is worth to note, that despite the impressive enhancements achieved by the memory optimizations introduced in the DWT (discussed in Subsection 3.5), the execution time is still dominated by the three vectorizable stages, leaving room for significant improvements in performance through SLP exploitation. In the following subsections we describe in detail how these three components were vectorized.

3.2. Irreversible Color Transform (ICT)

In the lossy compression process a classic forward inter-component transformation maps the data from the RGB to the YCbCr color space. It operates on all of the components together, and serves to reduce the correlation between components. The corresponding *for* loop nest that implements this linear transformation can be easily vectorized since it does not present dependencies and data items are sequential in memory. Spatial locality cannot be further improved and its nature make optimizations aimed at improving temporal locality ineffective, as no data reuse exist. However, this transformation was originally coded in *JasPer* using its own fixed-point arithmetic template class (a set of macros), which includes certain datatype conversions (*castings*) that inhibit automatic vectorization. On the other hand, some difficulties also arise in the vectorization of fixed-point computations. For instance, integer multiplication involves intermediate data twice as wide as the original operators, which prevents full SIMD exploitation. These difficulties lead us to perform a floating-point ICT transformation that produces exact agreement (within round-off error).

3.3. Discrete Wavelet Transform (DWT)

The algorithm used to compute the DWT in the JPEG2000 standard is the *Lifting* scheme [20].

Despite the inherent data parallelism of the DWT, *JasPer's* fixed-point arithmetic class, RAW dependencies between the different lifting steps, and strided memory accesses inhibit the automatic vectorization of this important component.

3.3.1. Vectorization of the Vertical filtering

Guided in part by our previous work [5], we have introduced the following optimizations into the *JasPer's* implementation of the DWT:

- *Loop fusion*: The *prediction*, *update* and *normalization* steps of the lifting [20] scheme have been fused in only one loop, enhancing temporal locality.

- *Loop interchange*. The spatial locality of the vertical filtering has been significantly improved using a loop-interchange transformation. The naïve implementation used in *JasPer* for this filtering suffers from an important memory bottleneck, given that data is processed by columns despite using a row-major layout. As a positive side-effect, this loop transformation moves the RAW dependencies to the outer loop, enabling the vectorization of this filtering [5].

- *Pipeline computation and array padding*. Array memory access has been further improved using *pipeline computation* [5], which enhances temporal locality, as well as using *array padding*, which reduces data cache conflict misses.

These optimizations, apart from improving memory hierarchy exploitation, allow for an efficient and straightforward vectorization of the vertical filtering performing the lifting steps of four consecutive columns in parallel. Among the different memory management alternatives we have employed the *inplace-mallat* strategy that we introduced in [6]. For the sake of conciseness, we refer to this previous paper for a more elaborate description.

3.3.2. Vectorization of the Horizontal filtering

Although *loop fusion*, *pipeline computation* and *array padding* also improve the performance of the horizontal filtering, its vectorization is a more challenging task. If data is processed row-by-row, which maximizes spatial locality, the inner loop of the horizontal filtering present RAW data dependencies that prevent vectorization. In contrast, if a loop interchange transformation is applied to move RAW dependencies out of the inner loop, strided memory access are necessary, degrading performance. Commercial compilers do not apply such transformations since their heuristics suggest that the vectorization benefits does not compensate the overheads caused by strided memory access pattern. A trade-off between memory access and inner-loop parallelism is necessary.

Transposing the whole array to allow for the same strategy employed in the vertical filtering is not feasible since the overhead of the transposition is larger than the benefits of vectorization. However, in [5] we found out that the combination of *pipeline computation* and a non-linear data layout, makes local transposition of small array tiles affordable. Unfortunately, we have ruled out this approach given that the integration of a non-linear layout within the JPEG2000, not only involves a significant coding effort, but also important overheads due to memory transfers in other stages of the coding process. Alternatively, we have adapted the methodology presented in [5] to a row-major data layout. The adaptation has been performed introducing the following optimizations in the horizontal filtering:

- The outer loop has been tiled so that several rows are processed simultaneously.

- The vector register file is used as a temporal buffer to hold four vectors loaded from consecutive rows. The vector block is then efficient transposed without needing additional memory accesses. To further improve memory access, the inner loop has been unrolled so that all vectorial loads are memory aligned.

- Temporal locality is further optimized storing the output vectors back to memory when they are no longer needed in the horizontal filtering, i.e. we perform a manual scalar replacement on vector data.

In summary, the core of our new strategy to vectorize the horizontal filtering is a local transposition that at the expense of additional data movements allows for performing the lifting steps of four consecutive rows in parallel, i.e. it enables the same strategy applied on the vertical filtering. Nevertheless, it is worth to note that without the memory optimization mentioned above, this transposition is quite inefficient.

3.4. Quantization

After transforming the image components, the real wavelet coefficients are quantized to an integer space. As in the ICT stage, the memory accesses in the corresponding loop nest cannot be further improved, but the vectorization is feasible since no data dependencies exist and data items are sequential in memory. However, the employment of the *JasPer's* fixed-point arithmetic class inhibit vectorization and we opted to perform intermediate computations using floating-point. In addition, we have removed the *if-statement* included in the original's quantization inner loop adding some extra arithmetic. This conditional statement, used to distinguish between positive and negative data, introduces unnecessary control dependencies that hamper performance. Our transformation not only makes vectorization possible but also enhances dramatically the performance of this stage, which already runs around 5 times faster than the original *JasPer's* quantization even without enabling vectorization.

3.5. Performance Results

We have reported separately in Tables 1 and 2 the speedups achieved by the memory-hierarchy and SLP optimizations. To isolate the different contributions to the overall speedup, these figures do not take into account the additional gains introduced by substituting *JasPer's* fixed-point arithmetic class and by removing *if-statements* in the quantization stage.

Table 1
Speedups achieved by the proposed memory optimizations. Local and Global denote the speedups on the DWT and how them translate into the whole compression process respectively.

Image Size	Local	Global
256x256	2.19	1.26
512x512	3.82	1.63
566x733	3.39	1.52
733x566	3.80	1.60
1024x1024	21.24	5.48
1038x1251	4.82	1.81
1251x1038	4.80	1.80
2048x2048	27.34	6.71
2341x2341	4.98	1.82
4096x4096	28.08	7.10

Table 2
Speedups achieved by SLP exploitation. ICT, DWT and QT stand for the local speedup on these stages while Global refers to the full compression process.

Image Size	ICT	DWT	QT	Global
256x256	3.51	1.66	3.97	2.95
512x512	2.31	1.95	3.64	2.45
566x733	1.84	1.72	3.43	2.17
733x566	1.83	1.84	3.52	2.20
1024x1024	1.76	2.08	3.70	1.96
1251x1038	1.66	1.84	3.44	1.82
1038x1251	1.67	1.89	3.66	1.85
2048x2048	1.76	2.20	3.68	1.75
2341x2341	1.65	1.96	3.38	1.64
4096x4096	1.77	2.26	3.72	1.72

In the DWT, the gains achieved through memory optimization increase with the image size as could be expected, although power of two image sizes represent a pathological case due to the impact of conflict cache misses, which are almost completely removed by means of array padding. The proposed vectorization delivers consistent speedups that range between 1.5 and 2.3 in the horizontal filtering and between 1.5 and 3.9 in the vertical one, which translates into an average overall speedup close to 2. The speedups achieved on the ICT and the Quatization stages are also consistent and very close to the ideal values.

Finally, Figure 3 shows the new execution time breakdown. The encoding stage becomes now the most demanding procedure and hence, further optimization efforts should be focused, directly or indirectly, on this stage.

4. Simultaneous Multithreading

In a previous work [22] we proposed an alternative approach to compute the DWT on SMT architectures based on a functional partitioning strategy. It showed to be more efficient than traditional data partitioning techniques, frequently used in shared memory multiprocessors, achieving an extra 30% of speedup. However, given that the DWT stage only accounts for around 15%-25% of the JPEG compression process after memory and SLP optimizations (see Figure 3), this strategy does not promise significant returns. Nevertheless, based on that approach, our intuition was that a functional partitioning of the whole compression process would likely be efficient. For color images, where different channels have to be processed, this partitioning can be performed using a pipeline strategy.

Given that our SMT architecture only allows for two simultaneous threads, we have split the lossy compression process into two stages. Taking into account load balancing issues, the first stage performs DWT and Quantization, whereas the second stage performs the encoding (tier1 and tier2).

Figure 4 shows the benefits of the proposed parallelization strategy on the target SMT platform.We have considered two parallel versions, with and without enabling SLP. The speedups are quite satisfactory (higher than 15% in most cases) taking into account both, the expected gains reported by Intel (around 30%) [14], and the parallel fraction of code ($2/3$). It is also worth to note that the performance improvements are higher when SLP is enable, which reproduces the behavior observed for the DWT in [22]. Figure 5 analyzes this synergy between SLP and SMT. The stripped bars show the overall speedup that could be expected by applying both SLP and SMT optimizations if both improvements were independent (i.e. assuming a multiplicative effect), whereas the black bars show the actual speedups. The latter speedups are always higher for all the image sizes.

Finally, Table 4 shows the impressive overall speedups achieved by our proposed implementation over the original *JasPer*. The overall gains range between 1.9 and 22 times depending on the image size.

5. Conclusions

The JPEG2000 is a well known image coding standard which is used nowadays in many multimedia applications and whose popularity will surely grow in the next few years. Its tuning for different platforms is becoming crucial for many imaging applications. Some promising sources for optimization are the exploitation of SLP and SMT.

Vectorization has been a hot topic since the introduction of vector supercomputers. However, when classical techniques are applied to SLP code generation, they can introduce a considerable overhead in the resulting codes. In this paper we have applied and extended a systematic approach

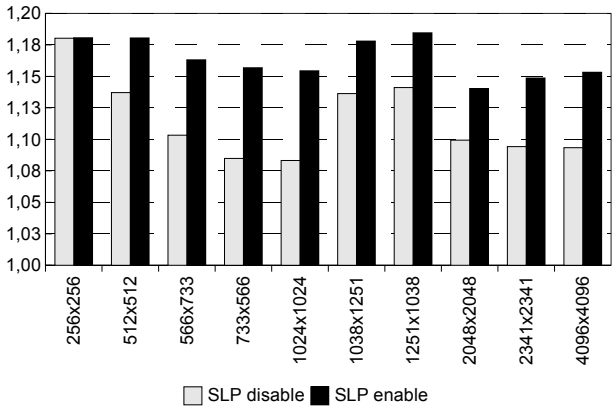


Figure 4. Speedup of the parallel version with and without turning on SLP.

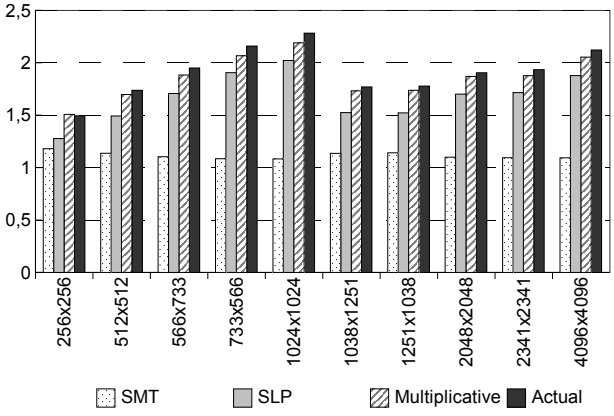


Figure 5. Synergy between SLT and SMT.

Table 3
Execution times (in milliseconds) of the JPEG2000 compression process. The column labeled as *Reference* refers to the original *JasPer*, whereas *Tuned* refers to our proposed implementation with all the optimizations turned on.

Image size	Reference(ms)	Tuned (ms)	SpeedUp
256x256	52	27	1,9
512x512	180	57	3,1
566x733	254	81	3,1
733x566	264	80	3,3
1024x1024	1993	157	12,6
1038x1251	836	192	4,3
1251x1038	828	188	4,4
2048x2048	9473	491	19,2
2341x2341	3820	642	5,3
4096x4096	39711	1780	22,3

applied in [22] to the DWT. Although hand-tuning has been needed to obtain satisfactory results, some general rules have been established to achieve an efficient implementation. Three stages in the lossy compression process have been successfully vectorized and our previous research about DWT vectorization has been extended with a new vectorization of the horizontal filtering based on a local transposition.

Furthermore, the compression process has been multi-threaded taking advantage of the functional parallelism available in multi-component images.

Overall, the performance of the *JasPer* implementation has been improved by a factor that ranges from 1.9 to 22, depending on the image size. In a future research we plan to apply this methodology to Motion JPEG2000, in an attempt to meet real-time video coding requirements.

References

[1] M. Adams and R. Ward. Jasper: A portable flexible open-source software tool kit for image coding/processing. In *IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*, 2004.

[2] Aart J.C. Bik, Milind Girkar, Paul M. Grey, and Xinmin Tian. Automatic intra-register vectorization for the intel architecture. *Int. J. of Parallel Programming*, 30(2):65–98, 2002.

[3] Doug Burger and James R. Goodman. Billion-transistor architectures: There and back again. *Computer*,

37(3):22–28, Mar. 2004.

[4] S. Chatterjee and C. D. Brooks. Cache-effi cient wavelet lifting in JPEG 2000. In *Proc. of the IEEE Int. Conf. on Multimedia and Expo*, pages 797–800, Lousanne, Switzerland, Aug. 2002.

[5] D. Chaver, C. Tenllado, L. Piñuel, M. Prieto, and F. Tirado. 2D wavelet transform enhancement on general-purpose microprocessors: Memory hierarchy and SIMD parallelism exploitation. In *Proc. of the 2002 Int. Conf. on High Performance Computing (HiPC'02)*, pages 9–21, India, Dec. 2002.

[6] D. Chaver, C. Tenllado, L. Piñuel, M. Prieto, and F. Tirado. Vectorization of the 2d wavelet lifting transform using SIMD extensions. In *Workshop on Parallel and Distributed Image Processing, Video Processing, and Multimedia (PDIVM'03)*, Nize, France, Apr. 2003.

[7] Keith Diefendorff, Pradeep K. Dubey, Ron Hochsprung, and Hunter Scales. AltiVec extension to PowerPC accelerates media processing. *IEEE Micro*, 20(2):85–95, 2000.

[8] The Portland Group. Pgi users guide : Parallel Fortran, C and C++ for scientists and engineers. Information available at http://www.pgroup.com/doc/pgiug.pdf.

[9] IBM Corporation. XL C/C++ advanced edition for linux. Information available at http://www-306.ibm.com/software/awdtools/xlcpp/features/linux/index.html.

[10] Intel. Intel C/C++ compilers for Linux. http://www.intel.com/software/products/compilers.

[11] Ronald N. Kalla, Balaram Sinharoy, and Joel M. Tendler. IBM Power5 chip: A dual-core multithreaded processor. 24(2):40–47, 2004.

[12] A. Khokhar, G. Hebe, P. Thulasiraman, and G. R. Gao. Load adaptive algorithms and implementations for the 2d discrete wavelet transform on fi ne-grain multithreaded architectures. In *Proc. of the Int. Parallel Processing Symp. (IPPS/SPDP 1999)*, Puerto Rico, Apr. 1999.

[13] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proc. of the ACM SIGPLAN 2000 Conf. on Programming language design and implementation (PLDI '00)*, pages 145–156, New York, NY, USA, 2000. ACM Press.

[14] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, J. Alan Miller, and Michael Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology J.*, 6(1):4–15, Feb. 2002.

[15] P. Meerwald, R. Norcen, and A. Uhl. Cache issues with JPEG2000 wavelet lifting. In *Proc. of 2002 Visual Communications and Image Processing (VCIP'02)*, pages 626–634, 2002.

[16] P. Meerwald, R. Norcen, and A. Uhl. Parallel JPEG2000 image coding on multiprocessors. In *Proc. of the Int. Parallel and Distributed Processing Symp.(IPDPS'02)*, Florida, Apr. 2002.

[17] Dorit Naishlos. Autovectorization in gcc. In *Proc. of the GCC Developers Summit*, pages 105–118, Ottawa, Canada, Jun. 2004.

[18] Gang Ren, Peng Wu, and David Padua. An empirical study on the vectorization of multimedia applications for multimedia extensions. In *Proc. of the 19th IEEE Int. Parallel and Distributed Processing Symp. (IPDPS05)*, Denver, USA, Apr. 2005.

[19] D. Santa-Cruz and T. Ebrahimi. A study of JPEG 2000 still image coding versus other standards. In *Proc. of the X European Signal Processing Conf.*, volume 2, pages 673–676, Finland, Sep. 2000.

[20] Wim Sweldens. The lifting scheme: A construction of second generation wavelets. *SIAM J. on Mathematical Analysis*, 29(2):511–546, 1998.

[21] S. Taubman and M. W. Marcellin. *JPEG2000: Image Compression Fundamentals, Standards, and Practice*. Kluwer Int. Series in Engineering and Computer Science, 2002.

[22] C. Tenllado, C. Garc´ıa, M. Prieto, L. Piñuel, and F. Tirado. Exploiting multilevel parallelism within modern microprocessors: DWT as a case study. In *Post-Conf. book of Vecpar'04*, pages 556–568, 2004.

[23] Shreekant (Ticky) Thakkar and Tom Huff. Internet streaming SIMD extensions. *Computer*, 32(12):26–34, 1999.

A Parallel IMAGE Processing Server for Distributed Applications

A. Clematis[a], D. D'Agostino[a], A. Galizia[a]

[a]IMATI-CNR, Genova, Italy

The use of parallel libraries for image processing is a common practice in the implementation of monolithic applications. Nowadays there is an increasing interest in moving towards distributed and heterogeneous applications also in the image processing community. The computational Grid may represent an adequate middleware support and infrastructure at this purpose. A problem of interest in this context is the interoperability and reuse of available parallel libraries, possibly through a dynamic interaction model that permits to accommodate request arriving from heterogeneous clients in a distributed environment. In this paper we present $\mathrm{PIMA(GE)}^2$, a Parallel IMAGE processing GEnova server obtained wrapping up a legacy code parallel library. The parallel server is implemented using CORBA and may be integrated in Grid architecture. The $\mathrm{PIMA(GE)}^2$ server organization and its interaction with user requests are discussed. Early experimental results are presented.

1. Introduction

In the era of distributed and Grid computing [6], the concept of library is evolving from a static software entity to a more dynamic one. A library is often considered as a static tool, mainly designed to be executed on an homogeneous architecture in order to develop monolithic applications. Moving towards distributed, dynamic and heterogeneous environments, it is transformed in a component-based bundle that can be accessed using a server based interaction. This new entity can be used to develop adaptive and distributed applications that are designed using a component based approach and executed in a changing and heterogeneous environment. It means that the functions performed by the library are evolved into components, and an application becomes a dynamic components concatenation, executed on an heterogeneous and distributed infrastructures. The server should be able to manage multiple requests performed by different distributed clients.
The integration of parallel libraries in heterogeneous and Grid-based environment is a topic of interest in the scientific community. The use of sequential and parallel libraries is well assessed for numerical applications [1], but also in other fields like image processing it is now possible to find efficient and effective libraries. These libraries permit to speed-up the development process and often provide optimized performances on a wide range of target architectures. Their quality can be assessed using a well defined set of requirements that permits to verify the library effectiveness and efficiency. These requirements include:

- easy of use, the most compelling need for the users is a simple and natural tool to exploit the operations allowed, therefore a good point is represented by a well designed interface;
- transparency to the parallelism and the optimization policy, it is essential to provide tools that shield their users from the intrinsic complexities of parallel computations;
- efficiency, the users should be capable of obtaining significant performance gains in the most common image processing operations;
- portability, it is essential to ensure executions on different target machines, especially with the new emerging technologies and architecture;

- robustness, it is essential to provide computations that are insensitive to the variations of the data and ensure correct results;
- scalability, a well-designed software architecture has to increase its performance under the different used technologies, especially when resources are added;
- completeness, the users have not the necessity of using other packages related with the environment or the parallelism.

Considering the level of quality and the development cost of image processing parallel libraries it is a great interest their interoperability and reuse in distributed and heterogeneous environments. At this purpose in the recent past different technologies and methodologies have been developed in order to obtain interoperability and reuse of legacy code, one of the best suited methodology is code encapsulation [16], while a suitable technology is represented by the CORBA framework [18].
Our goal is to permit interoperability and reuse of a parallel image processing library in an heterogeneous environment obtaining at the same time a high level of flexibility and a dynamic behaviour permitting to develop Client Server image processing applications that exploit performance figures provided by an optimized parallel library and efficiently utilize the underlying distributed infrastructure.
To achieve this goal we have designed and implemented $\mathrm{PIMA(GE)^2}$, a Parallel IMAGE Processing GEnova Server [2], obtained by wrapping an image processing legacy code using CORBA [3]. The computational kernel is provided by a C++/MPI-2 parallel library, exploiting the algorithmic patterns that characterize image processing operations. This library is considered, for the purpose of the definition of the server as a legacy code. In order to permit code encapsulation an adequate object hierarchy is adopted without any actual change to the legacy code; its aim is to drive the definition of an adequate CORBA $\mathrm{PIMA(GE)^2}$ Application Programming Interface (API). One the most important problem in the design of the internal $\mathrm{PIMA(GE)^2}$ structure is the embedding of C++/MPI-2 computations within a CORBA object with a limited overhead. The focus of the paper is on this aspect and the adopted design solutions and early experimental results are presented. It is important to notice that our solution is based on standard CORBA implementation, according to the Object Management Group, (OMG) [11], specification.
Different related works addresses similar goals. In the field of image processing an increasing attention is paid to parallel processing. In fact we find different and actual examples of parallel library, for example VSIPL++ [9], ParHorus [15], PIPT [12]; they provide object-oriented image processing code, and ensure high performance executions. They are compliant with the requirements defined in the list presented above in this Section. The use of CORBA in order to wrap parallel applications has been considered in different works [4], [13], [5]; it is obtained through the definition of CORBA compliant parallel objects and environments. Finally the problem of library integration in distributed and Grid based architectures has been considered by different projects like Netsolve [10] and GrADS [8] .
In this paper we shortly outline the library interface and then we provide a more detailed description of the $\mathrm{PIMA(GE)^2}$ internal organization and behaviour.

2. Parallel image processing library structure

The main goal of the $\mathrm{PIMA(GE)^2}$ server is the encapsulation of C++/MPI-2 parallel library in order to use it in a distributed environment. The library implements a large set of the most common image processing operations, according to the classification and the organization provided in Image

Algebra, [14]. The computation is data parallel, with a coordinator process during the initialization and the I/O phases; the parallelism is transparent to the user and totally managed by the library. Also the optimization policy applied in the library is transparent; they are both hidden through a sequential API. The optimization policy is oriented to different levels: to perform an opportune management of communication and memory operations; data distribution is aimed to obtain load balancing among the MPI processes. The library itself is an ongoing work, in fact other efforts are spent to obtain an adaptive code [7]; we are interested in designing code that operate efficiently on different target architectures.
Focusing on the provided operations of the library, we can group them in conceptual objects, that are not intended to prescribe how an operation is performed but to underline the operation similarity and to help in the definition of an effective and efficient interface. The conceptual objects allow an easier management of the library operations, since the user is no longer involved with a large number of functions, but he has to consider and handle a small set of clear and well-defined objects. The operations are grouped together according to different rules, such as the nature similarity, or the data structures processed. In this way we outlined in the library code eight objects: I/O Operations, Point Operations, Image Arithmetic, Reduce Operations, Geometric Operations, Neighborhood Operations, Morphology Operations, Differential Operations. We considered a hidden code level, it is not included in the library code classification; the library hidden code is concerned with the management of secondary data structures, with the parallelism and the optimization policy.

3. The library hierarchy and the $PIMA(GE)^2$ IDL interface

To allow a simple and coherent use of the library in a distributed environment, the most relevant element is the definition of effective and flexible interfaces, that permit the development of efficient image processing applications. It can be achieved through a natural evolution from the library legacy code to the $\mathrm{PIMA(GE)}^2$ server; for this reason we exploit the conceptual model introduced in the previous Section, and define a logical hierarchy of image processing operations and objects. Its organization is presented in Figure 1. The aim of the object hierarchy is to drive the definition of an adequate CORBA interface, and its Interface Description Language (IDL) based implementation. It represents the $\mathrm{PIMA(GE)}^2$ Application Programming Interface (API).

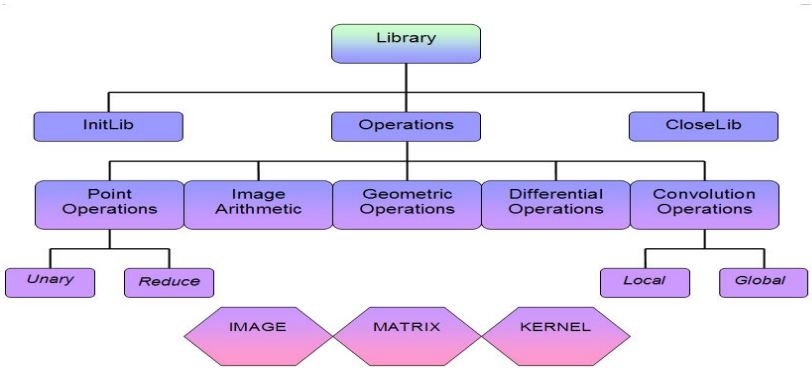


Figure 1. Overview of the $\mathrm{PIMA(GE)}^2$ hierarchy

3.1. The basic elements of the hierarchy

The basic elements of the hierarchy are represented by the main data structures of the library: IMAGE, used to store images; KERNEL, used to store convolution kernels; and MATRIX, used to store geometric matrices. We relate to them functions for memory management, I/O operations, and generic operations when it is possible, for example to transpose or invert a MATRIX. In this way we obtain three objects representing the first layer of the $PIMA(GE)^2$ object hierarchy; they already include, as methods, some of the objects derived by the library classification and part of the hidden library code level.

3.2. The image processing objects

The core of the $PIMA(GE)^2$ server is a set of five objects, obtained by grouping together the library objects, according to the rules already mentioned:

1. **Point Operations** Operations that take in input only one IMAGE object, i.e. square root, absolute value, sine, or a collective value, i.e. the maximum, minimum pixel value of an IMAGE;

2. **Image Arithmetic** Operations that take in input two IMAGE objects, i.e. addition, product, etc;

3. **Geometric Operations** Operations that take in input one IMAGE object and one MATRIX object, i.e. rotation, translation and scaling;

4. **Convolution Operations** Operations that take in input one IMAGE object and one KERNEL object, for example involving a neighborhood of each pixel: percentile, median. or combined together by two binary functions: Gaussian convolution, erosion, dilation.

5. **Differential Operations** Operations that take in input one IMAGE and perform differential operators, i.e. Hessian, gradient, Laplacian.

4. The server internal structure

The main difficulty in the server design is due to the presence of parallelism in the computation. It imposes the management of two different environments: the CORBA framework (server side) and the MPI library (legacy code side). They are aimed to manage different kind of problems, therefore there is not a standard schema to allow their cooperation. In fact CORBA was born to develop mainly sequential applications, hence it does not support intrinsic compatibility with any kind of parallel environment. We decided to use a standard CORBA specification, TAO [17] , and look for a way allowing the execution of the parallel computation inside a CORBA object.
To achieve this goal we designated a process to perform specific tasks. We took advantage from the presence of the coordinator MPI process during the initialization and I/O phases; this process acts as the gateway that coordinates the two environments, allowing a cooperative computation. It has a dual position in the software architecture, in fact it will be at the same time a CORBA and a MPI process. On the CORBA side, it has to activate the ORB and to perform as a CORBA server; at the same time it is also one of the spawned MPI processes, and it has to manage the parallel computation. It performs as an MPI process that is able to interact with the CORBA environment, collecting the client requests, and managing the parallel computation in a properly way, i.e. invoking the services required from the clients with their proper parameters. This solution is possible because, acting as a CORBA server, the designate process knows the Client requirements and is able to communicate

them to the others MPI processes. In this way it coordinates the computation and gives back the information to the Clients. This solution does not require modifications to the legacy code, since it exploits the presence of the coordinator MPI process during the initialization and I/O phases. The behavior is presented in the Figure 2.

Another important feature of the $\mathrm{PIMA(GE)}^2$ server organization is concerned with data man-

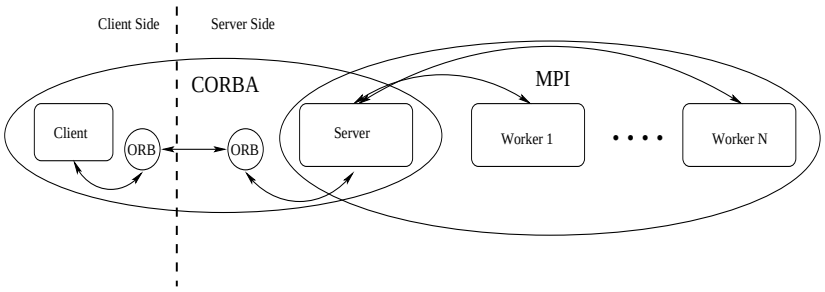


Figure 2. The CORBA-MPI interaction

agement. In order to minimize the overhead introduced in the wrapping phase, a good point is to improve the exploitation of the environment through the development of services oriented to the data transfer and smart strategies in the communication channel management. A considerable improvement in the computations is brought if data are managed by the most proper tools, as an example large size data transfer across a distributed environment can be more efficiently accomplished using optimized FTP protocols instead of the ORB channel. Providing optimized data transfer services also leads to a clear separation between movement, access, and processing of data. This point represents a key element in the exploitation of a distributed architecture.

At this aim we remark that an application calls a pipeline of library functions acting on the same data, with the same degree of parallelism. This remark permits to assume that a sequence of operations required by the client, may be performed entirely on the server on the set of images that are distributed once and kept on the parallel nodes till the end of the pipeline, thus avoiding useless movement of large data through the ORB channel. In order to force this type of interaction between the client and the server, we implemented the server in such a way that:

- The server executes a sequence of operations issued by the client as some kind of atomic action, in the sense that the parallel context and data distribution to parallel nodes remain unchanged during the whole sequence;
- The client-server interaction inside a sequence of image operations is limited to the exchange of symbolic identifier of images.

It is important to notice that these implementation strategy does not introduce heavy dependencies in the client code. The only aspect that the client has to take in consideration is that it has to interact with a CORBA server. An example is provided in the next Section.

5. Client-Server interaction example code

The Client code looks like that of a process that uses standard CORBA service and standard library routines; in fact the policy applied to allow MPI-CORBA compatibility is totally hidden to the

Client and managed by $\mathrm{PIMA(GE)}^2$ server. The optimization policy is embedded in the server code as well, and also the use of the symbolic identifiers is managed by the server and it is not perceived by the clients.
As it happens using a specific domain library, it is necessary to invoke the initialization and closure services. Nothing more is required. To perform a selected operation on an image, the Client has to relate an integer value to it, with a normal declaration; in the example code below it is the only one performed. The declared integer identifies its unique correspondent image during all the Client-Server session, and it corresponds to the symbolic identifier used to minimize the use of the ORB channel.
The example of Client code shows the transparence and easy of use of $\mathrm{PIMA(GE)}^2$ services.

Client code

```
int main(int argc, char* argv[]) {

 CORBA::ORB_var orb = CORBA::ORB_init (argc, argv,"");
 std::string ior; std::ifstream is(argv[1]); std::getline(is, ior);
 CORBA::Object_var obj = orb->string_to_object(ior.c_str());

//get the pimage object reference
 LIB::Operazioni_var pimage = LIB::Operazioni::_narrow(obj.in());

//declaration of the integer value used to identify an imag
 CORBA::Long origImg, resImg;

 pimage->InitLib();

//implicit association of the integer origImg to the image
//stored in the file "myimg"
 pimage->readImageFromFile_C("myimg",origImg);
 pimage->getdimension(origImg,imWidth, imHeight, imDepth);

//implicit association of the integer resImg to a new image
 pimage->createImage_C(imWidth, imHeight,imDepth, resImg);

 pimage->rotImg(IdImg, resImg, 60, LINEAR, 0, p);
 pimage->reflectImg(resImg,doX,doY);
 pimage->writeImageToFile_C(resImg,"myresult");
 pimage->deleteImage_C(origImg);
 pimage->deleteImage_C(resImg);
 pimage->CloseLib();

 orb->destroy();

 return 0; }
```

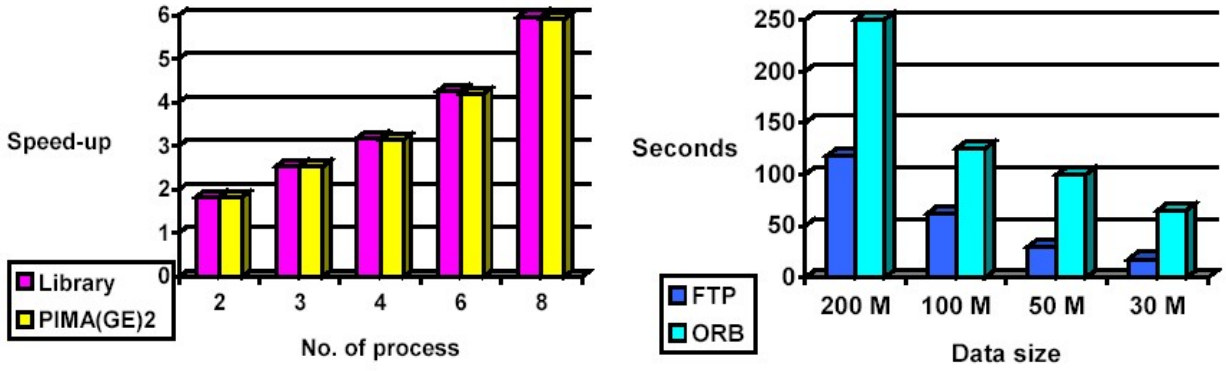
6. Experimental results

The experimental results we obtained show a reasonable overhead due to the presence of the CORBA framework. We carried out tests aimed to assess the performance of the $\mathrm{PIMA(GE)}^2$ server compared with the parallel library, and to evaluate the benefit obtained by minimizing the use of the ORB communication channel, during the invocation of a $\mathrm{PIMA(GE)}^2$ services sequence, and in order to transfer data from the client to the server. The application used for the tests is aimed to

the detection of linear structure in an image. In all the cases the parallel execution environment is a Linux cluster with eight nodes interconnected by a Gigabit switched Ethernet, and each node is equipped with a 2.66 GHz Pentium processor, 512 Mbytes of RAM and two EIDE disks interface in RAID 0. The client is a PC that is located on the same local area network, in order to be able to evaluate overheads that are due to our software infrastructure and limit other possible sources of overheads.

The tests assert that the CORBA set-up time is a fixed value (about 1-2 seconds in our experiments), and does not depend on the number of parallel MPI processes. When the interaction between the client and the server is going to grow, we have a performance degradation due to a more consistent use of the ORB communication channel; but applying the symbolic identifiers strategy the overhead is quite limited especially for compute intensive applications. Using symbolic identifiers is a strategy that leads to a fairly unvaried speed-up performance. Comparing the speed up of the application performed with the library functions and one obtained implementing the application with the $\mathrm{PIMA(GE)}^2$ services we obtained that the variation of the speed-up is around 1.3%; experimental results are plotted in Figure 3(a).

From the data transfer point of view, we prove that definitely better performance are obtained exploiting FTP protocols; in fact the execution time using the ORB channel is about twice of the one obtainend using the ftp protocol. This behavior is indipendent from the data size, according to the results presented in Figure 3(b).



(a) Speed-up comparisons

(b) Data transfer services comparisons

Figure 3. Graphics reporting the comparison between the speed-up of the library and $\mathrm{PIMA(GE)}^2$, the left one, and the comparison between the data services, the right one

7. Conclusion

An approach to high performance legacy code encapsulation in a grid-oriented environment is presented; we described a way to reuse an MPI-based parallel image processing library in distributed environment, using CORBA without modifications to the OMG standard. Also the legacy code had no modifications. To allow the cohesistence of CORBA and MPI environment, we exploit

the presence of a the coordinator MPI process during the initialization and I/O phases. In order to reduce the overhead introduced in the wrapping, the $\mathrm{PIMA(GE)}^2$ server architecture has been improved with more specific data transfer services. This step also allowed the use of an optimized ORB channel management strategy.

8. Acknowledgments

This work has been supported by MIUR programme L.449/97-00 High Performance Distributed Computing Platform, and by FIRB strategic project on Enabling Technologies for Information Society, Grid.it.

References

[1] R.F. Boisvert: Mathematical Software: Past, Present and Future. Computational Science, Mathematics, and Software, Purdue University Press, 2002.
[2] A. Clematis, D. D'Agostino and A. Galizia: An Object Interface for Interoperability of Image Processing Parallel Library in a Distributed Environment. In Proceedings of ICIAP 2005, LNCS 3617, pp. 584-5891, 2005.
[3] The CORBA home page, http://www.corba.org/
[4] A. Dennis, C. Peréz and T. Priol: Portable parallel CORBA objects: an approach to combine parallel and distributed programming for Grid Computing. Euro-Par 2001
[5] A. Dennis, C. Peréz and T. Priol: PadicoTM: An open integration framework for communication middleware and runtimes. Future Generation Computer System, 19-4. May 2003
[6] I. Forster and C. Kesselman: The grid: blueprint for a new computing infrastructure, 2nd Edition. Morgan Kaufmann Publishers Inc. 2004.
[7] A. Galizia: Evaluation of optimization policies in the implementation of Parallel Libraries. Technical Report IMATI-CNR-Ge, N. 20/2004
[8] GrADS Project, Grid Application Development Software Projects, http://www.hipersoft.rice.edu/grads/
[9] J. Lebak, J. Kepner, H. Hoffmann and E. Rudtledge: Parallel VSILPL++: an open standard library for high-performance parallel signal processing. IEEE Proceedings, vol. 93-2. February 2005
[10] NetSolve Project, http://icl.cs.utk.edu/netsolve/
[11] OMG Official Website, http://omg.org
[12] Parallel Image Processing Toolkit Home Page, http://www.osl.iu.edu/research/pipt/
[13] C. Peréz, T. Priol and A. Ribes: A Parallel CORBA Objects for programming Computational Grids. Distributed Systems Online, 4-2. February 2003
[14] G. Ritter and J.Wilson: Handbook of Computer Vision Algorithms in Image Algebra, 2nd edition. CRC Press, Inc. 2001
[15] F. Seinstra, D. Koelma and J.M Geusebroek: A software architecture for user transparent parallel image processing. Parallel Computing, 28. 2002
[16] H.M. Sneed: Encapsulation of Legacy Software: A technique for reuse software components. Annals of Software Engineering, vol. 9. 2000
[17] Tao home page, http://www.cs.wustl.edu/schmidt/TAO.html
[18] N. Wang, D.C. Schmitdt and D. Levine: An overview of the CORBA component model. Addison-Wesley. 2000.

A Speculative Parallel Algorithm for Self-Organizing Maps*

C. García[a], M. Prieto[a], A. Pascual-Montano[a]

[a]Dpto. de Arquitectura de Computadores y Automática
Universidad Complutense, 28040 Madrid, Spain
e-mail: {garsanca, mpmatias}@dacya.ucm.es, pascual@fis.ucm.es

We have explored in this paper the parallel implementation of *Kohonen's Self Organizing Map* (SOM) in simultaneous multithreading architectures. A new method is proposed that outperforms classic *map-partitioning* approaches targeted for shared-memory multiprocessors. As an application study we have chosen an image classification problem in 3D Electron Microscopy. Performance results are taken using real biological data on an *hyperthreading*-enabled Intel Pentium 4.

1. Introduction

Electron Microscopy (EM) is a valuable tool for the elucidation of the three-dimensional structure of macromolecular complexes [3]. However, it faces different methodological problems. Most of the methods used for 3D reconstruction in EM rely on the strict requirement that the individual projection images considered for the reconstruction process, correspond to different views of the same biological specimen. The achievement of such a set of particles makes necessary a preprocessing step aimed at sorting the original population of images into different homogeneous sub-populations. This classification is also difficult since in most of the instances no prior information on the macromolecule structure is available. Furthermore, the large amount of electron microscopy projection images needed for a single study (usually tens of thousands), makes its computational efficiency another major concern for researches.

In this paper we have addressed this efficiency issue, focusing on simultaneous multithreading (SMT) processors and the well-known *Self-Organizing Map* (SOM) algorithm [9]. SOM was introduced in the EM field by Marabini and Carazo in [12], although more recently new variants were proposed [15–17].

As its name suggests, SMT allows several independent threads to issue instructions simultaneously in a single cycle [20]. Its main goal is to yield better use of the processor's resources, hiding the inefficiencies caused by long operational latencies. Some sort of SMT (denoted as *hyperthreading*) has been already incorporated into the Intel's Pentium 4 and Xeon families [13] as well as the IBM's Power5 [7], being expected to become ubiquitous soon in most superscalar processors. Despite this architectural trend, the exploitation of this capability is often limited by the relative underdevelopment of the compilers (i.e. automatic parallelization).

This noticeable divergence between compiler technology and computing power has forced us to explicitly adapt SOM to take advantage of the simultaneous thread-issue capability. At first glance, SMT processors can be seen as a set of logical processors that share some resources. Consequently, one may think that optimizations targeted for symmetric multiprocessors systems (SMP) are also good candidates for SMT. However, we will show that this naïve view does not provides satisfactory speedups.

*This work has been supported by the Spanish research grants TIC 2002-750 and GR/SAL/0653/2004. A.P.M. also acknowledges the support of the Spanish Ramon y Cajal Program.

As a baseline code to study SMT-aware optimizations, we have employed a modified version of the well-known SOM-PAK package from the Helsinki University of Technology [18]. This tuned version already includes some code rearrangement that allows for an efficient exploitation of the Intel's SSE and SSE2 media extensions. Experiments have been performed on an Intel Pentium 4 running at 3.2 Ghz (1MB L2 cache, 1Gb DDR400). Thread synchronization has been performed using POSIX Threads.

The rest of this paper is organized as follows: Section 2 introduces SOM and summarizes its most popular parallel implementations; Section 3 describes the application problems used for validation and assessment. Section 4 presents our alternative implementation and its performance. Finally the paper ends with some conclusions.

2. Self-Organizing Maps

The *Self-Organizing Map* (SOM) is an unsupervised neural network that provides a non-linear mapping from a high-dimensional input space to a low-dimensional output space (most often a 2D output grid). Its simplicity in description and practical implementation, have made SOM one of the most popular and widely used methods for pattern recognition and exploratory data analysis [8]. Many studies have also confirmed its ability in producing an orderly mapping of high dimensional data items onto a regular low dimensional grid. Its main property is that it quite consistently conserves the original topological and metric relationships of the items.

As show in Figure 1, the SOM consists of a set of *i* input units, corresponding to the input data set, and a set of *j* output units arranged in a two-dimensional regular grid with a predefined topology. Each output unit has a codevector $w_i \epsilon \Re^p$ associated with it.

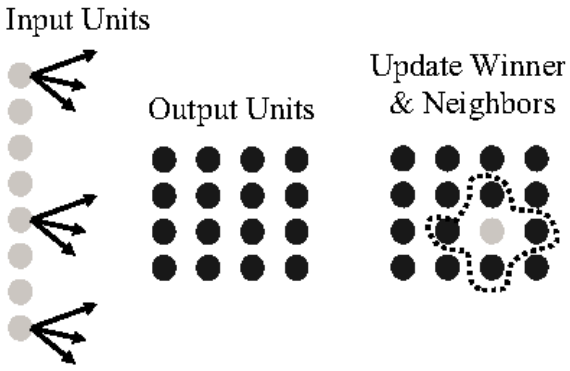


Figure 1. Self-Organizing Maps.

The functionality of the algorithm can be described as follows: when an input vector ($x_i \epsilon \Re^p$) is presented to the net, (1) the neurons or units in the output layer compete with each other and (2) the winner (which is that neuron whose codevector has the minimum distance from x_i) as well as a predefined set of its neighbors, update their values. This process is repeated until some stopping criterion is met, usually, when the codevectors stabilize or when a certain number of iterations are completed. The update rule for the output vectors used in this algorithm can be mathematically described as:

$$w_k(t+1) = w_k(t) + \alpha(t) h_{ck}(t)[x(t) - w_k(t)] \quad (1)$$

Where $\alpha(t)$ is a decreasing function of t (time, or iteration number) that controls the magnitude of the changes with time, and $h_{ck}(t)$ is a function that controls the size of the neighborhood of the winning node to be updated during training. Both $\alpha(t)$ and $h_{ck}(t)$ decrease monotonically during training in order to achieve convergence. This classic algorithm is usually denoted as *on-line*, given that it updates the codevectors at each step [6].

2.1. Conventional Parallel Self-Organizing Maps

A significant amount of work on the parallel implementation of SOM has been performed in recent years [10,1,14,11]. In general, two different alternatives, which we have denoted as *map-partitioning* (MP) and *data-partitioning* (DP), have been explored.

In *map-partitioning*, the SOM is distributed among the different threads so that every thread processes every input data and trains its share of the map. This approach preserves the ordering of updates shown in Eq. (1) and hence produces exact agreement (within round-off error) with the *on-line* algorithm described above. Its main drawback is its high synchronization cost: threads should synchronize at each step after determining the winning output unit, and again after updating the map. This overhead only makes MP attractive for hardware implementations, and for parallel architectures with low latency synchronization primitives such as *Shared Memory Symmetric Multiprocessors* (SMP) or SMT processors.

In *data-partitioning*, input data are distributed across threads so that every thread trains a full copy of the map using a subset of the input space. Its granularity is coarser than the small-grained parallelism available in MP, which decreases synchronization costs. Unfortunately DP does not fit with the *on-line* update established by Eq. (1) and hence it should be combined with different variants of SOM. A typical choice is *batch* SOM [9,6,10,11]. However, *batch* SOM is not very popular in the scientific community because of two main reasons:

1. It requires all data items to be present, which is impractical in some applications due to communication time or disk space requirements.

2. It is more difficult to obtain a properly ordered map than with the classical *on-line* algorithm [6].

Taking both factors into account, and given that a SMT processor provides for efficient thread synchronization, we have used a MP implementation of the classic *on-line* SOM as a point of reference for evaluating the advantages of our alternative.

3. Description of the application problems

The validation and assessment of our method has been performed using three different real datasets corresponding to three different macromolecular studies:

- *MCM*. In this study we used 4723 single particles (64x64 projection images) from the MCM helicase from Methanobacterium Thermoautotrophicum obtained by electron microscopy [5]. The aim of SOM is to study the structural heterogeneity of this macromolecule in an attempt to understand its biological function. Each 64x64 image was arranged in a vector resulting in a 4723x4096 data matrix. Both the serial and the map-partitioning versions of the *on-line* SOM algorithms were applied to this dataset using a 30x10 hexagonal grid.

- *G40P*. This test consists of 2120 EM images corresponding to 2D projections of negatively stained hexamers of the bacteriophage SPP1 G40P helicase [2]. From each image, 1580 pixels

within an area of interest were extracted using a binary mask. The aim of SOM in this case, which is computed using a 2120x1580 data matrix, is to study the structural polymorphism of this macromolecular assembly.

- *Tomograms*. To extend our study to a more demanding case, we also tested our method using 3D images from electron tomography. The problem in this case is focused on classification of three-dimensional volumes, which represent a much larger and complex problem. The dataset used for testing contained 1000 3D images from the insect flight muscle specimen. The tomograms were properly carved out and aligned from the tomographic map as described in [21]. As in *G40P*, only a specific area of interest was extracted using a proper binary mask producing 1000 vectors with 19475 components (voxels) each. More details about this dataset can be found in [21,17].

4. Speculative Parallel SOM

4.1. Motivation

Rows labeled MP in Table 1 show the speedups achieved by *Map-Partitioning*. We also report the speedups achieved on a dual Intel's Xeon server to emphasize the discrepancies between SMT processors and SMP systems. As mentioned above, the baseline code refers to our hand-tuned version of SOM-PAK, in which the computation of distances between input and output units as well as the update of codevectors, have been vectorized using the intrinsic function interface provide by the Intel C/C++ compiler [4]. Vectorization already achieves a significant speedup that ranges between 3 and 4,depending on the size of the images and the number of items.

As expected, MP provides satisfactory results on SMP platforms. However, the corresponding performance gains achieved on SMT are negligible, either using a block or a cyclic distribution of output units across threads.

The main reason behind this poor behavior is the competition among threads for the shared resources, especially for the largest dataset (*Tomograms*). In particular, we have observed that given that both threads process different output units, they must compete for the memory bandwidth and share data caches. Driven by these discouraging results we have devised an alternative way to perform the computation, in which competition among threads for the data caches is much lower.

4.2. Description

Our alternative algorithm is based on the following observations:

1. In the first steps of training, the neighborhood of the winning node of a certain input vector usually overlaps with the neighborhoods of the winning nodes of consecutive input vectors.

2. Given that $h_{ct}(t)$, i.e. the neighborhood radius, decreases monotonically during training, the chances of overlapping also tends to decrease.

3. If the neighborhoods of two consecutive winning nodes do not overlap, it would be possible to process the respective input units in parallel.

Our proposed algorithm tries to exploit the potential parallelism described by the third observation. It uses a master-slave scheme (see Figure 2), in which the master thread basically follows the conventional *online* SOM algorithm, and the slave or helper thread tries to train the map with the successive input units in a speculative way. After determining its winner node (both threads perform

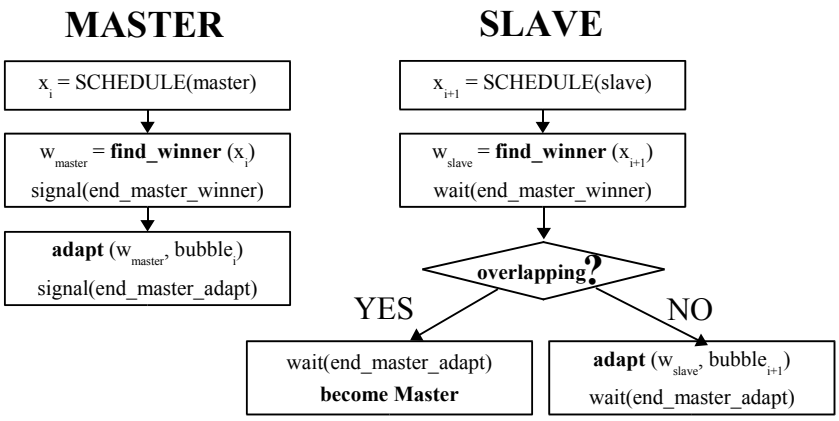


Figure 2. Master-slave scheme.

the *find_winner* stage of the algorithm in parallel), the slave checks if its respective neighborhood overlaps with the master's neighborhood. If they do overlap, the speculation has been wrong and the slave must wait until the end of the master's update phase, and then master and slave exchange their roles. The new slave tries speculatively to process another input vector. If they do not overlap, we assume that both, the master and the slave threads, can update the map concurrently and proceed with the next input vectors.

Since the chances of overlapping are high at the beginning and tend to decrease during training, speculation starts after processing a certain number of input-vectors. Based on our experiments (see Section 4.3), the length of this *warm-up* phase has been set to the first ten percent of the iterations. *Thread parallelism* is also exploited in this initial phase applying MP.

The obvious disadvantage of speculation is that some resources are allocated to useless computations that must be re-executed. However, since resources are often underutilized on current microprocessors [20], the benefit of speculation could far outweigh this disadvantage. It is also worth to note that this scheme can be scaled using more slaves, and sorting them out so that the overlapping checking is performed in order. Nevertheless, within current design trends (current SMT processors only allow for two simultaneous threads), this possibility has no practical interest.

4.3. Performance Results

Rows labeled SP in Table 1 show the speedups achieved by our alternative approach. Obviously, in the dual system its performance is significantly worse than MP but in the SMT processor, it far outperforms MP if vectorization is enabled (baseline code). The results are quite satisfactory taking into account the achievable speedups reported by Intel (around 30% on average)[19]. Hardware performance counters have confirmed that this better behavior is due to its more efficient exploitation of the memory hierarchy. Unlike MP, where threads compete for the data caches and the memory bandwidth, the master and the slave threads tend to cooperate during the *find_winner* stage, since they analyze each of the output units approximately at the same time. This way, temporal locality is better exploited. We should highlight that in the SP model, a certain synergy between SMT and vectorization exists. Vectorization allows a better exploitation of the shared resources and allows for a better cooperation amongst threads. It is also worth to note that the overall speedup of our new implementation over the original SOM-PAK version is close to 5 (on average).

Figures 3-5 show the evolution of the misspeculation rate (the percentage of overlaps found during

Table 1
Speedups achieved by a conventional *Map-Partitioning* decomposition (MP) and the proposed *Speculative* implementation (SP) on a SMT processor and a dual processor server (denoted as SMP) respectively for the three different databases considered.

MCM	SMT	SMP
Original-MP	1.03	1.77
Baseline-MP	1.07	1.59
Original-SP	1.01	1.46
Baseline-SP	1.24	1.08

G40P	SMT	SMP
Original-MP	1.04	1.91
Baseline-MP	1.09	1.89
Original-SP	1.02	1.41
Baseline-SP	1.23	1.12

Tomograms	SMT	SMP
Original-MP	1.04	1.68
Baseline-MP	1.00	1.17
Original-SP	1.01	1.48
Baseline-SP	1.11	1.06

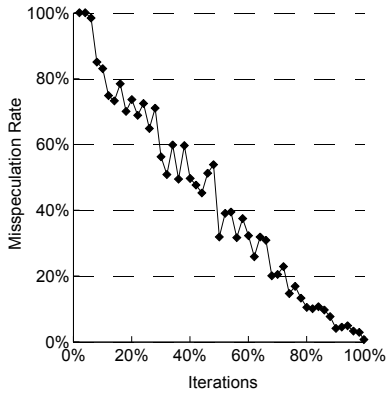


Figure 3. Evolution of the misspeculation rate for the MCM dataset.

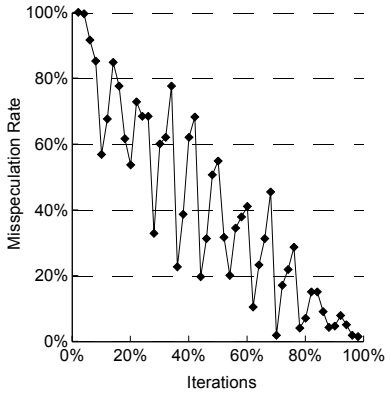


Figure 4. Evolution of the misspeculation rate for the G40P dataset.

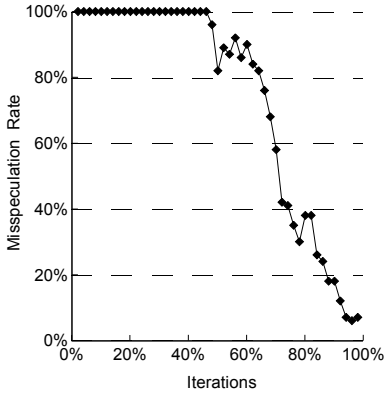


Figure 5. Evolution of the misspeculation rate for the Tomograms dataset.

execution) using the SP model without a *warm-up* phase (i.e. applying speculation from the first iteration). They explain the worse behavior observed on the *Tomagrams* dataset since the number the overlaps keeps high for a large number of iterations. Increasing the warm-up length in this case does not improve performance significantly since it only removes the misspeculation overheads. In fact, these overheads are not very significant since misspeculation also causes data-prefetching as positive side effect.

4.4. Validation

Figure 6 compares the results obtained by the classic *online* SOM algorithm and our speculative approach using as input the *MCM* dataset. The numbers in the lower right corner of each unit represent the number of original images assigned to each code vector in the map. It is visually evident that both maps are almost identical, which represents a qualitative evidence that the speculative approach do not affect the overall numerical output of the map. The correlation coefficient of both maps yielded a 0.9999 similarity. The small differences between both results are only reflected in the number of images represented by each of the code vectors in those maps. In some cases, a few very similar images are assigned to different neighboring units. This is not a problem whatsoever due to the fact that SOMs represents a smooth representation of the input data in the 2D ordered grid and therefore, a cluster of images is determined by a set of neighboring units and not by individuals.

The resulting maps obtained with the *G40P* dataset showed a correlation coefficient of 0.9981 and, again, this difference is only reflected in the location of some similar images that were allocated to neighboring units in the map. Results demonstrate that even if our *Speculative* calculation of SOMs

might introduce some small changes in the code vectors, the overall results are not affected.

Finally, in the case of tomograms dataset, the resulting maps using the *online* and the *speculative* SOMs were identical (correlation coefficient of 1).

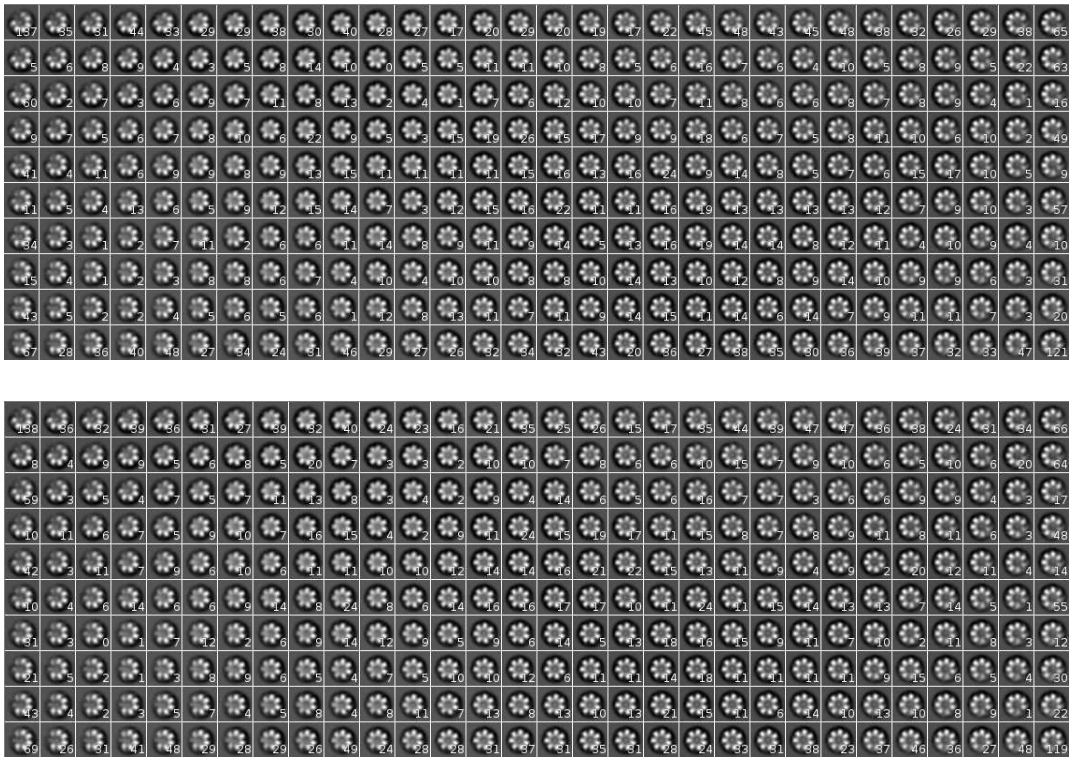


Figure 6. 30x10 Self-Organizing Map obtained using the *online* Kohonen Map (top) and the proposed *speculative* Kohonen Map (bottom).

5. Conclusions

The self-organising map is a popular unsupervised neural network model for high-dimensional data analysis. However, the high execution times required to train the map put a limit to its use in many application domains, where either very large datasets are encountered and/or real-time response times are required.

We have introduced a new parallel implementation of the classic *Self-Organizing Map* algorithm, which on SMT processors fits better than traditional *Map-Partitioning* strategies due to a better exploitation of the memory hierarchy. It is also worth to note the numeric results achieved with this speculative approach are almost identical to those obtained with the serial *online* SOM.

The importance of this contribution is justified by the high popularity of this method in the data analysis process in life sciences and technology. Our speculative implementation, which also includes explicit usage of Intel's SSE and SSE2 media extensions, achieves an impressive speedup over the original SOM-PAK (close 5 on average). This performance adds a valuable extra benefit in this process, allowing scientist and researchers to analyze their data nearly interactively.

References

[1] D. Merkl A. Rauber, P. Tomisch. parSOM: a parallel implementation of the self organizing map exploiting cache effectsmaking the SOM fit for interactive high-performance data analysis. In *Proc. of the IEEE-INNS-ENNS Int. Joint Conf. on Neural Networks*, volume 6, pages 177–182, 2000.

[2] M. Barcena, C. S. Martin, F. Weise, S. Ayora, J. C. Alonso, and J. M. Carazo. Polymorphic quaternary organization of the bacillus subtilis bacteriophage SPP1 replicative helicase (G40 P). *J Mol Biol*, 283:809–819, 1998.

[3] W. Baumeister and A. C. Steven. Macromolecular electron microscopy in the era of structural genomics. *Trends Biochem Sci*, 25:624–31, 2000.

[4] Intel Corpation. Intel C/C++ and Intel Fortran Compilers for Linux. Available at http://www.intel.com/software/products/compilers.

[5] Y. Gomez-Llorente, R. J. Fletcher, X. S. Chen, J. M. Carazo, and C. San Martin. Polymorphism and double hexamer structure in the archaeal helicase mcm from methanobacterium thermoautotrophicum. *Submitted*, 2005.

[6] M. Cottrell J.C. Fort, P. Letremy. Advantages and drawbacks of the batch kohonen algorithm. In *10th European Symp. On Artificial Neural Networks*, Bruges (Belgium), 2005.

[7] Ronald N. Kalla, Balaram Sinharoy, and Joel M. Tendler. IBM Power5 chip: A dual-core multithreaded processor. *IEEE Micro*, 24(2):40–47, 2004.

[8] S. Kaski, J. Kangas, and T. Kohonen. Bibliography of self-organizing map (SOM) papers:1981–1997. 1:102–350, 1998.

[9] Teuvo Kohonen. Self-organized formation of topologically correct feature maps. pages 509–521, 1988.

[10] R. D. Lawrence, G. S. Almasi, and H. E. Rushmeier. A scalable parallel algorithm for self-organizing maps with applications to sparse data mining problems. *Data Min. Knowl. Discov.*, 3(2):171–195, 1999.

[11] F. Luengo, A.S. Cofi no, and J.M. Gutierrez. GRID oriented implementation of self-organizing maps for data mining in meteorology, 2004.

[12] R. Marabini and J. M. Carazo. Pattern recognition and classification of images of biological macromolecules using artificial neural networks. *Biophys J*, 66:1804–1814, 1994.

[13] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, J. Alan Miller, and Michael Upton. Hyper-threading technology architecture and microarchitecture. *INTEL-TECH-J*, 6(1):4–15, February 2002.

[14] H. Y. Ming and N Ahuja. A data partition method for parallel self-organizing map. In *Proc. of the IEEE Int. Joint Conf. on Neural Networks 1999 (IJCNN 99)*, volume 3, pages 1929–33, 1999.

[15] R. D. Pascual-Marqui, A. Pascual-Montano, K. Kochi, and J. M. Carazo. Smoothly distributed fuzzy c-means: a new self-organizing map. *Pattern Recognition*, 34:2395–2402, 2001.

[16] A. Pascual-Montano, L. E. Donate, M. Valle, M. Barcena, R. D. Pascual-Marqui, and J. M. Carazo. A novel neural network technique for analysis and classification of EM single-particle images. *J Struct Biol*, 133:233–245, 2001.

[17] A. Pascual-Montano, K. A. Taylor, H. Winkler, R. D. Pascual-Marqui, and J. M. Carazo. Quantitative self-organizing maps for clustering electron tomograms. *J Struct Biol*, 138:114–122, 2002.

[18] Kohonen T., Hynninen J., Kangas J., and Laaksonen J. SOM PAK: The self-organizing map program package. Technical Report Technical Report A31, Helsinki University of Technology, Laboratory of Computer and Information Science, 1996.

[19] X. Tian, A. Bik, M Girkar, P. Grey, H. Saito, and E. Su. Intel OpenMP C++/Fortran compiler for hyper-threading technology: Implementation and performance. *Intel Technology Journal*, 6(1), 2002.

[20] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *25 Years ISCA: Retrospectives and Reprints*, pages 533–544, 1998.

[21] H. Winkler and K. A. Taylor. Multivariate statistical analysis of three-dimensional cross-bridge motifs in insect flight muscle. *Ultramicroscopy*, 77:141–152, 1999.

Genomic-scale analysis of DNA Words of Arbitrary Length by Parallel Computation*

X.Y. Yang[a], A. Ripoll[a], V. Arnau[b], I. Marín[b], E. Luque[a]

[a]ETSE, Universitat Autònoma de Barcelona Computer Science Department 08193-Bellaterra, Barcelona, Spain

[b]Universidad de Valencia Campus de Burjassot Avd. Vicent Andrés Estellés,s/n. 46100 Burjassot, Valencia

In the post-genomic era, one of the main tasks is deciphering the meaning of the DNA sequences of complex organisms. In order to do so, there is a clear need for biocomputer tools able to extract and order the information of long DNA molecules, such as whole chromosomes or even complete genomes. However, most genomic analyses have been concentrated on the detection and counting of short words having sizes of between 1 and 10 nucleotides. In this paper, we describe parallel algorithms with different complexities that exhaustively determine all words of size k, k being arbitrarily large, in a source DNA sequence. The results shown that our algorithms achieve a high degree of scalability, allowing the detection of DNA words of 64 nucleotides in only 800 seconds.

1. Introduction

In genomic analysis, the determination of the DNA words (sequence of nucleotides) found in chromosomes or even of whole genomes is essential in many contexts. Some examples are: 1) Determination of genomic signatures that characterize organisms or species; 2) Characterization of differences among chromosomes for certain specific oligonucleotides, such as the differences in CpG dinucleotides that are specific targets for DNA methylation in many organisms; 3) Characterization of repetitive DNA sequences; or, 4) Finding singular sequences that are much more frequent in certain chromosomes or genomes than in others. However, many of these analyses become unfeasible for long word lengths on standard PC equipment due to memory limitations.

Significant advances in our understanding of genome structure and complexity have been provided by the analysis of oligonucleotide words (reviewed in [7]). However, most of the analyses performed to date have been limited to the detection and counting of short words (of sizes k between 1 and 10 nucleotides; $1 \leq k \leq 10$) [1][8]. When those analyses are to be extended to longer words, the complex problem emerges of designing algorithms that are able to handle the millions of possible combinations (4^k; i.e. for k = 15, the number of possible words is about 10^9). One solution is to use complex data pre-processing ([6]) that introduces a high computational requirement.

In this study, we propose two parallel algorithms [3][4][5] for the DNA words frequency analysis: *two*-stage and *k*-stage algorithm. The *two*-stage algorithm is able to achieve high parallel computer-system efficiency to perform DNA exhaustive analysis, extracting the frequency information of every word of a particular length in a DNA sequence. The *k*-stage algorithm is designed for highly frequent DNA word analysis, such as Alu sequences. The *k*-stage is able to find frequency information about extremely long DNA words. In our study, we evaluated our algorithms with human chromosome analysis using a cluster of 16 PCs. Our results show that *two*-stage algorithm achieves up to 76% of

*This work was supported by the MCyT-Spain under contract TIC 2004-03388, TIC 2003-08154-C06-04 and partially supported by the Generalitat de Catalunya- Grup de Recerca Consolidat 2001SGR-00218.

the cluster optimal performance and the *k*-stage algorithm is able to analyze DNA words longer than 64 nucleotides.

The remainder of this paper is organized as follows: we show the sequential algorithm definition in section 2. The *two*-stage and *k*-stage parallel algorithms are analyzed in sections 3 and 4, respectively. In section 5, we show performance evaluation and indicate our main conclusions are set out in section 6.

2. The Sequential Algorithm

In a previous study [2], we described an algorithm that can exhaustively determine all 12-nucleotide-long ($k = 12$) words present in a given DNA sequence, together with their frequencies.

The rationale of the algorithm is as follows: a tree is started that has a root node (level 0) from which four different pointers can be established, corresponding to nucleotides A, C, G or T, that lead to the four possible nodes in the level 1. In the level 2 of the tree, we have 16 nodes, corresponding to 16 different words with 2-nucleotide (AA, AC,...,TT). The solution tree structure is dynamically generated to contain all the posible words. To build the tree at a faster rate, a pointer was used for each level, so that after reading one nucleotide, each pointer indicates a new node determined by the pointer on the previous level and the newly read nucleotide. The final nodes have a different data structure. They have a counter that indicates the frequency of appearances of each word of *k* nucleotides.

In that study, we also showed that the algorithm is fast enough to be used on a genomic scale. However, the algorithm is not able to extend the frequency analysis to DNA sequences with more than 12 nucleotides. Given a DNA sequence that is long enough to contain all the possible combinations of words of length *k*, the maximum number of nodes (N_{max}) of the solutions tree of the sequential algorithm is:

$$N_{max} = \sum_{i=1}^{k} 4^i = \frac{4^{k+1} - 1}{3} \tag{1}$$

In the implementation, the sequential algorithm requires a value of 32 bits (4 bytes) to reference a node of the tree. Since each node makes 4 references to nodes of the next level of the tree, each node requires, at least, 16 bytes. In a 32-bit machine, the maximum amount of available memory is 2^{32} bytes. In other words, it is able to hold a tree of $\frac{2^{32}}{16} = 2^{28} = 4^{14}$ nodes. This implies that the sequential algorithm would not be able to solve the problem of searches for words of longer than 13 nucleotides using a 32-bit machine.

Moreover, if the machine has less than 2^{32} bytes (4 Gigabytes) of memory, the solvable space for the problem, using the sequence algorithm, is even smaller. For example, if we have 512 Mbytes of memory, the sequential algorithm is only able to deal with a search problem for words of up to 12 in length. One Gigabyte of memory is not enough to extend the search to a length of 13. The result of this analysis leads us to the unquestionable need to research an algorithm using parallel systems where there is more than one computational element.

3. *Two*-stage Parallel Algorithm

In this section, we show the key ideas of the parallel algorithm. The aim of this algorithm is to be able to extend the initial problem to analyse words of any length.

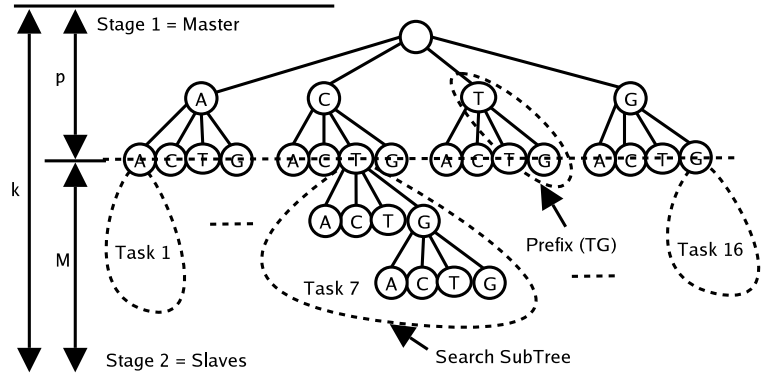


Figure 1. Parallel Task Assignation

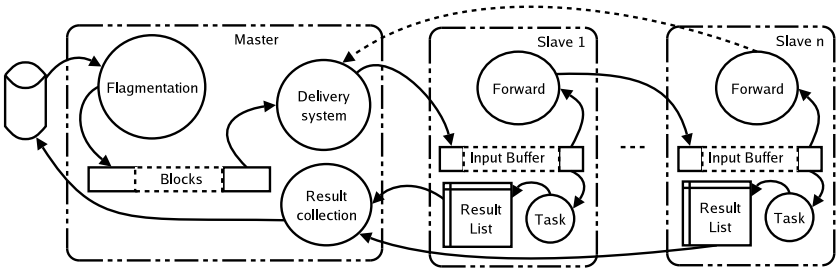


Figure 2. DNA information Delivery Mechanism

3.1. Definition of Parallel Tasks

The number of levels of solution tree depends on the length of the words that we want to analyze and is limited to a maximum (M) that is determined by the size of the memory of the machine being used. The key idea of the parallel algorithm consists of dividing the process of generating the search tree into two stages in which a partial tree of only the first p levels is generated for stage one (Figure 1). Stage 2 performs the search process for all of the sub-trees that are derived from the leaf nodes of the partial tree in stage 1. The levels of the sub-trees in stage 2 will be limited by M and therefore the process can be performed on a single machine. Each of these sub-trees represents one task that should be performed in stage 2 and the number of these is equal to 4^p. The value of p depends on M and k that is the length of the words we want to search for; so that $p = k - M$. Figure 1 shows an example of a 4-levels tree for searching for words of length 4 ($k = 4$). In this case, the value of M is 2 and, therefore, p is 2.

Since the sub-trees are derived from different leaf nodes of the partial tree in stage 1, the tasks related to these in stage 2 are independent from each other and, therefore, can be run on parallel machines following the Master-Slave model. In stage 1, the master generates the partial tree, which defines different tasks of the stage 2. Each path between the root node and a leaf node of the partial tree defines the *Prefix* of the words that should be analyzed by a task. The slaves generate the sub-trees in stage 2 and represent the different tasks of analysis. Depending on the number of machines available for the parallel system, the master creates a number of slaves and sends them the different prefixes to determine the analysis task they must perform. In the case of Figure 1, the master generates a partial tree of 2 levels and defines the 16 tasks. The 16 tasks are defined by 16 prefixes that are sent from the master to the slaves. For example, the task 7 searches for the frequency of all the words that start with a $\{CT\}$ prefix.

3.2. DNA sequence Delivery and Results Collection Mechanism

As well as the distribution of the task prefixes to slaves, the master sends the DNA sequence to be used for an analysis. Since every slave requires the complete DNA sequence in order to generate the sub-tree, the master have to send the DNA sequence to all of the slaves. An initial solution could be for the master to send the sequence to the slaves using independent communication channels. The major problem with this solution is that it quickly saturates the system's communication network and the parallel algorithm cannot be scaled to every number of machines.

Our design for the DNA sequence delivery is based on the logical chaining of slaves (Figure 2). The master assigns two neighbours to each slave; one neighbour that collects the DNA sequence (V_s) and one neighbour that sends the sequence (V_d). All of the DNA sequence that has been collected from V_s is redirected towards V_d (except the last slave) using the Forward mechanism. In this way, DNA sequence is sent to every slave.

Algorithm 1 Slave Pseudocode

```
1: Receive Control Information from Master Prefix ⇐ {P_1, P_2, ..., P_p}, K and m
2: Create a array of (K − m) pointers P[0..K − m] ⇐ {NULL, .., NULL}
3: Create a array of (K − m) numbers Pos[0..K − m] ⇐ {1, .., 1}
4: while There is more DNA information do
5:    Receive one Nucleotide(NN = 1, 2, 3, 4) from Mastar
6:    for i = 1 to (K − m) do
7:       if Pos[i] == m then
8:          P[i] ⇐ Next level of Tree acording to N
9:          if P[i] == NULL then
10:            Create next level of the Tree
11:         end if
12:         if P[i] is the last level then
13:            Increase the Frequency
14:            P[i] ⇐ NULL
15:            Pos[i] ⇐ 1
16:         end if
17:      else
18:         if NN == Prefix[Pos[i]] then
19:            Pos[i] ⇐ Pos[i] + 1
20:         end if
21:      end if
22:   end for
23: end while
```

The chaining mechanism works as a Pipeline process, where the stages of the pipeline represent the slaves that perform the searching task and re-send the DNA information. The performance of the pipeline closely depends on the DNS sequence delivery mechanism, which has to guarantee information availability as well as no overflow of the input buffers. The DNA sequence delivery is complemented by the control mechanism in which a message is sent by the last slave (slave n) to the master to report the successive arrivals of packets of information. Using the information on the frequency of arrival of these messages, the master adapts its delivery speed to the processing capacity of the different stages.

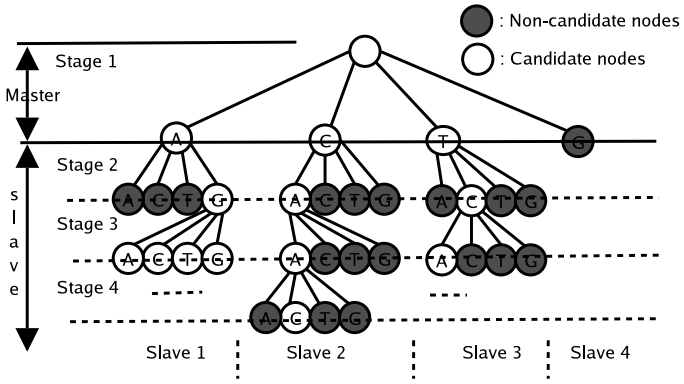


Figure 3. *k*-stage Algorithm Solution Tree

In order to deliver DNA sequence on a packet commutation based network, the master divides the DNA into blocks. This is performed in the Fragmentation module. As well as the division, the module also performs the DNA sequence compacting process by assigning 2 bits to each base. As a result of the analysis, the slave generates a list that contains the words found with a frequency higher than the threshold. The slaves' lists are collected by the master to generate the final output of the system.

Algorithm 1 show the slave pseudocodes. In the step 1, the slave receives the control information from the master whileas the DNA sequence is received in step 5. $P[i]$'s are memory pointers to scan the tree and $Pos[i]$'s indicate the position of the DNA word that each pointer is. The master pseudocode is not shown for space limitation.

4. *K*-Stage Parallel Algorithm with Dynamic Memory Deallocation

In the solution process of *two*-stage algorithm, a distributed tree is created that includes every word in the input DNA sequence, independently of frequency of appearance. But should we only wish to search for very frequently repeated words, the system does not use memory efficiently, as we will be storing information about uninterested words. For example, in the chromosome 1 of Homo sapiens, there are more than 40000 words with a frequency higher than 200, if the words-length is 12 ($k = 12$), However, less than 10000 of these appear at a frequency of more than 600. These results show that the number of words of interest depends on the threshold frequency and the word length. The higher the threshold frequency, the lower the number of words found. The longer the words are, the fewer there will be that will achieve the threshold frequency.

The aim of the *k*-stage parallel algorithm is to achieve better performance in the use of memory and number of machines. The key idea of this algorithm is to perform a progressive and directed analysis. In analysis process, only those parts of the tree that contain significant words (i.e. above the cut-off value) are created.

4.1. Definition of the *k*-stage Algorithm

The initial problem of searching for nucleotide words can be formulated mathematically as the search for all words of $\{b_1, b_2, ..., b_k\}$ such that $f(\{b_1, b_2, ..., b_k\}) > U$ where $f()$ is the function that determines the frequency at which any word appears. Given this formulation, the searching problem follows the axiom: *given a word* $\{b_1, ..., b_{k-1}, b_k\}$ *with* $f(\{b_1, ..., b_{k-1}, b_k\}) > U$ *if and only if* $f(\{b_1, ...b_{k-1}\}) > U$. This axiom tells us that if a word of length k is more frequent than U, then the word formed by the first $k-1$ nucleotides should also have a frequency greater than U.

The *k*-stage parallel algorithm is based on the aforementioned axiom in which each stage *i* searches

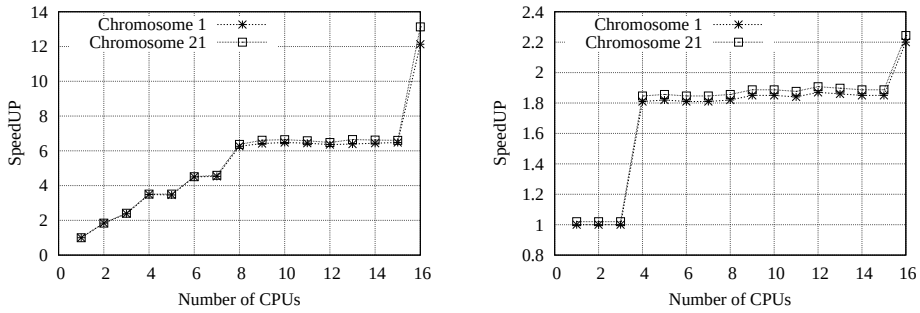


Figure 4. SpeedUP of: a) *two*-stage Algorithm. b) *k*-stage Algorithm.

for the frequency of words of length i and eliminates all those that are below threshold U. The tree of stage i is used to create the tree of stage $i+1$. In this way, only the nodes that contain possible solutions are kept in the memory, achieving higher memory efficiency. Figure 3 represents the tree of solutions created in accordance with a *k*-stage algorithm. The number of stages is 4 in this case, where stage 1 is run by the master and the other 3 in the four slaves. Two types of nodes have been defined: 1) candidate nodes, which are those that could contain solutions. 2) non-candidate nodes, which are those with a frequency lower than U and where therefore there are no longer any possible solutions in these branches of the tree. In the case shown, a single word {C,A,A,C} has been found by slave 2 that has a higher frequency than the threshold. In the case of slave 4, no branch has been created.

5. Performance Results

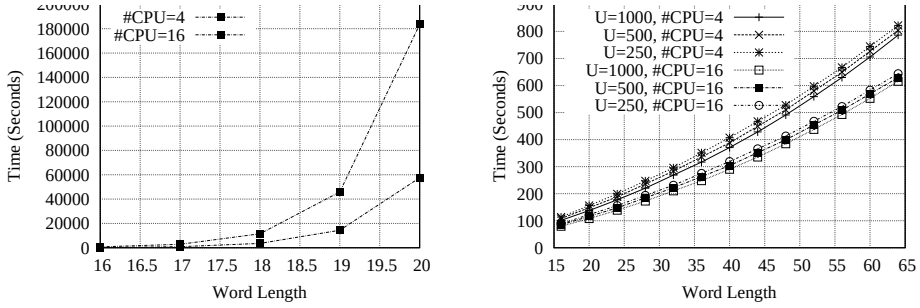
In this section, we describes the experimental results obtained by running the different versions of the algorithm using the parallel system. The different parallel algorithms are implemented using language C with the C+PVM library and the parallel system consists of a cluster of 16 Pentium 4 PCs with 512 MB of memory interconnected using a 100 Mbps Ethernet. There are several points that we are especially interested in measuring: 1) the speed-up that can be obtained using parallel algorithms. 2) the response time of parallel algorithms given a certain number of machines. 3) how the response time of the algorithms varies when we add more machines to the parallel system.

5.1. Speed-up of two- and *k*-stage Algorithms

We calculated the time taken to search for words of length 14 with a cut-off frequency of 1000. We ran the programs that implement the 2- and *k*-stage parallel algorithms on 1-16 machines. The M value of our machines is 12 and therefore in the case of *two*-stages, we found that $p = 14-12 = 2$ and therefore, the program must run $4^2 = 16$ independent tasks.

Figure 4.a shows the speed-up results in the cases of human chromosomes 1 and 21 using the *two*-stage algorithm. The speedup value increases as more machines are added to the running of the program. The algorithm obtains a speedup of 12.13 with 16 CPUs, which is 76% of the theoretical value.

Between 8 and 15 CPUs, no increase of the speed-up value is observed owing to the existence of non-usage of the CPU in the task distribution process. For example, in the case of 12 CPUs, the algorithm assigns 12 tasks to 12 CPUs in the first distribution, leaving 4 tasks to be run. In the next distribution, the system runs the 4 remaining tasks using only 4 CPUs and there are therefore 8 CPUs that do not do any task.



a) b)

Figure 5. a) Response time of *two*-stage. b) Response time of *k*-stage.

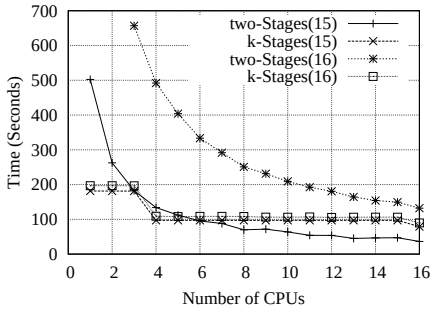


Figure 6. Response time Comparison according with Number of CPUs

Figure 4.b shows the speed-up values using the *k*-stage algorithm. In this case, the speed-up value is lower than the case for *two*-stages. This is because multiplying the number of machines by 4 involves the reduction of one stage of calculation. With 4 machines, we only achieve a speed-up of 1.85 and with 16 CPUs, the speed-up value is only 2.3.

5.2. Response Time According With the Number of Machines

We measured the computation time of 2- and *k*-stage algorithms using 4 machines to perform a search for words of lengths between $16 \leq k \leq 64$ and with a frequency of appearance greater than 250, 500 and 1000.

Figure 5.a shows the response time in seconds using the *two*-stage algorithm. In this case, we have only determined the time for words of a length of up to 20. The time required increases exponentially in accordance with the length. No outstanding differentiations are observed when varying cut-off frequency (U).

Figure 5.b shows the times taken in seconds by the *k*-stage algorithm. As we can see, the response time of *k*-stage algorithm increases linearly with value of *k*. There is also a slight increase depending on the value of the cut-off frequency (U). This is because with a lower cut-off value, there are more words in the final result that should be collected by the master.

5.3. Algoristhms Response Time Comparison

In order to make a comparison of the performance of the two versions of algorithm, we made a search for words of lengths 15 and 16 with a frequency of appearance greater than 500. In this test, we determined the total time of the two algorithms using from 1 to 16 CPUs.

Figure 6 shows what results were found. Owing to the complexity of the *two*-stage algorithm, the time increases exponentially as the number of available machines decreases. Meanwhile, the *k*-stage algorithm lineally increases the total time as fewer CPUs are used.

There is a point where the *two*-stage algorithm achieves better results than the *k*-stage algorithm.

This is the case for words of length 15. The *two*-stage algorithm achieves better results when there are more than five CPUs in the system. However, the result is not the same in the case of the search for words of 16, where the *k*-stage algorithm is always better than the *two*-stage one when we do not have more than 16 machines in the system.

From observing the data, the point can be calculated where the *two*-stage algorithm is better than the *k*-stage one. The following expression estimates the number of CPUs where the *two*-stage algorithm betters the *k*-stage one:

$$\frac{4^{k-M}}{N} \leq k - \log_4 N \tag{2}$$

For example, if we want to find chains of 15 using machines that have $M = 12$, the *two*-stage algorithm is better than the *k*-stage one when we have about 5 CPUs. In the case for 16, we can estimate that from around 19 machines, the *two*-stage algorithm is better than the *k*-stage one with regards to response time.

6. Conclusions

In this study, we present a parallel application that enables us to determine the frequency of appearance of words of *k* nucleotides in long DNA sequences. The proposed parallel algorithm has demonstrated high scalability in accordance with the number of processors and can be used to analyse any length of words in the search problem.

The proposed parallel algorithm presents two implementations. In the first algorithm, the distributed tree of solutions is totally built to obtain an exhaustive analysis. In the second implementation, however, the algorithm eliminate no intereted words and only the most frequently repeated words are presented in the solution tree. The second design achieves a high memory efficiency for analysis of extremely large words.

In order to validate the parallel algorithms, a set of tests was performed on various human chromosomes and the most repeated sequences found were highly repetitive sequences typical of our genome, known as Alu sequences plus simple DNA sequences(e.g. poli(A), poli(GA), etc). Those results were as expected when high cut-off values are used. Similar analyses can be performed on less known genomes in order to establish the most frequent highly repetitive sequences found in these.

References

[1] A.J.Gentles and S.Karlin. Genome-scale compositional comparisons in eukaryotes. *Genome Res. 11*, pages 540–546, 2001.

[2] V. Arnau and I. Marín. Fast algorithm for the exhaustive analysis of 12-nucleotide-long dna sequences. applications to human genomics evolution. In *Proceedings of the 17th IPDPS-HiCOMB 2003, Nice (France)*, 2003.

[3] R. Buyya. *High Performance Cluster Computing (Vols. 1 y 2)*. 1999.

[4] J. C. Cunha, P. Kacsuk, and S. C. W. Nova. *Parallel Program Development For Cluster Computing: Methodology, Tools and Integrated Environments.* 2001.

[5] A. G. et alter. Addison Wesley. *Introduction to Parallel Computing, Second Edition*. 2003.

[6] J. Healy, E. Thomas, J. Schwartz, and M. Wigler. Annotating large genomes with exact word matches. *Genome Res. 13*, pages 2306–2315, 2003.

[7] S. Karlin, A. M. Campbell, and J. Mrázek. Comparative dna analysis across diverse genomes. *Annu. Rev. Genet. 32*, pages 185–225, 1998.

[8] R. Roset, S. J.A, and X. Messeguer. MREPATT: detection and analysis of exact consecutive repeats in genomic sequences. *Bioinformatics*, 19 no. 18:2475–2476, 2003.

Parallel implementation of SEMPHY-a structural EM algorithm for phylogenetic reconstruction

Eric Li[a], Zhengqing Ouyang[a], Xi Deng[b], Yimin Zhang[a], Wenguang Chen[b]

[a]Intel China Research Center, Kexueyuan South Road #2, Haidian District, Beijing 100080, China

[b]Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China

Inference of phylogenetic trees based on the Maximal Likelihood method is computational extremely intensive. In order to accelerate the computations, we present the parallel implementation of SEMPHY in this paper. SEMPHY is a program for the phylogenetic reconstruction from DNA/protein sequence data, which uses the structural expectation-maximization (SEM) algorithm, to efficiently search for maximum likelihood phylogenetic trees. It is dramatically faster than other ML methods while reserving comparable accuracy. Two different parallel paradigms, i.e., OpenMP and MPI version are developed to compare their performance on different parallel systems. Our experiments show that the OpenMP version scales well with the increase of processors on the shared memory system, and achieves better speedup than the MPI version on the cluster system.

1. Introduction

Phylogenetic tree represents the phylogenetic relationship of species by a tree where closely related species are placed in nearby branches. For DNA/protein sequences from different species, a phylogenetic relationship among them can be inferred to reflect the course of evolution. The phylogeny of genes or species is broadly used in gene family classification, species divergence time estimation and disease associated mutation identification, etc [1].

Phylogenetic tree inference is a high performance computing problem along with the exponential increasing of biological data [2]. The inference includes searching for the best tree topology and the best branch lengths representing the distance between the two neighbors. If we have n taxa, the number of possible rooted trees is $(2n-3)!/2^{(n-2)}(n-2)!$ and that of unrooted trees is $(2n-5)!/2^{(n-3)}(n-3)!$ [3]. Another intrinsic complexity is the numerous branch length probabilities for each topology. The search for the best tree is a NP-complete problem [4]. The exhaustive search for the whole tree space is prohibitive except that the number of taxa is small, say 12, which evaluates approximately 6.5×10^8 possible trees and takes approximately two hours on a Pentium III machine by a moderate fast tree building algorithm [5]. Currently many ongoing phylogeny practices involve tens to hundred of taxa which is unimaginable for exhaustive tree search in a reasonable time scale. Extremely, the Ribosomal Database Project-II consists of 108,781 16S rRNA sequences to be analyzed for phylogeny (RDP) [6], and the Tree of Life project aims to illustrate the evolutionary tree that unites all living things (ToL) [7]. The availability of large data sets poses a great challenge on how to both accurately and efficiently reconstruct phylogenetic trees.

There are many approaches to construct the phylogenetic tree, such as the Neighbor Joining, Maximum Parsimony, etc [8], where the maximal likelihood (ML) approach is one of the most accurate methods [9]. However, the ML method is limited in relative small data set for its huge computational intensity, even after incorporating heuristic search techniques. For example, fastaDNAml [10] takes roughly 9 days for a single run on a data set containing 150 DNA sequences of each length 1269 characters and typically tens to thousands of different runs are needed because of its heuristic search

nature. For protein sequences, the problem of ML tree inference is even more demanding because the alphabet size of amino acids is twenty comparing to four of nucleotides.

Recently a new algorithm called SEMPHY is proposed for learning ML trees [11]. The main idea of SEMPHY is to use the structural expectation-maximization algorithm [12, 13], a variation of the EM algorithm for structure learning, to efficiently search for maximum likelihood phylogenetic trees. This algorithm is dramatically faster than other ML approaches while reserving comparable accuracy.

In this paper we present SEMPHY parallel implementation and analysis in both share and distributed memory environments. The parallel version of SEMPHY makes the algorithm more powerful for handling much more large data sets with hundreds of taxa, which covers most size range of current phylogenetic analysis. Particularly, it is the first time phylogenetic tree inference for large protein data set (up to 300 taxa) becomes realizable.

This paper is organized as follows. Section 2 describes the SEMPHY algorithm. Section 3 presents the analysis and parallel implementation of the SEMPHY algorithm in different parallel environments. Section 4 shows the experimental results. Section 5 concludes the paper.

2. Algorithm Description

The maximal likelihood (ML) method is a well-established statistical method of parameter estimation. The ML method of phylogenetic inference attempts to reconstruct a phylogenetic tree using an explicit evolution model. Mathematically speaking, given data D and a proper nucleotide (or protein) substitution model, ML methods will find a tree T, t such that $P(D|T,t)$ is maximized, where T and t represent the tree topology and the branch lengths respectively.

Exhaustive search of tree space (including the topology space and parameter space) that maximizes the log-likelihood of tree (T,t) becomes impractical when the number of taxa increases. Heuristic techniques are usually used to reduce the computation time, including the stepwise addition algorithm fastDNAml and the star-decomposition algorithm in MOLPHY [14], etc. Even using heuristic approaches, these methods still suffer from intensive computations. The structural EM algorithm used in SEMPHY efficiently solves this problem by using parameters found in current topology to help evaluate new topologies and thus improving the structure at each step until convergence. This is accomplished by a decomposition step of likelihood function.

X. Ma et al. [7] show that the expected log-likelihood for an arbitrary tree can be decomposed as the sum of local terms plus a constant:

$$Q(T,t) = \sum_{(i,j)\in T} l_{local}(E[S_{i,j}|D,T^0,t^0],t_{i,j}) + const \tag{1}$$

Where $E[S_{i,j}(a,b)]$ is the expected count of co-occurrences of the pair (a,b) in each position of (i,j):

$$E[S_{i,j}(a,b)] = E[\sum_{m=1}^{M} 1\{x_i[m]=a, x_j[m]=b\}] \tag{2}$$

Thus the Q score can be optimized by optimizing each l_{local} term separately within each iteration of the structural EM algorithm.

2.1. Structural EM algorithm

The structural EM is a variation of standard EM algorithm for learning structures. The general framework is similar to the standard EM procedure except that it optimizes not only the edge length

but also the topology during each EM iteration. In the phylogenetic inference, the algorithm consists of the following steps:

E-step: Compute $E[S_{i,j}(a,b)|D,T^l,t^l]$ for all links of (i,j), and for all character states $(a,b) \in \sum$.

M-step I: Optimize link length by computing $t_{i,j}^{l+1} = argmax_t l_{local}(E[S_{i,j}|D,T^l,t^l],t)$. Compute $w_{i,j}^{l+1} = l_{local}(E[S_{i,j}|D,T^l,t^l],t^{l+1})$. Denote $W^{l+1} = (w_{i,j}^{l+1})$ to be the $2N-2$ by $2N-2$ matrix weights.

M-step II: Finding the spanning tree T_*^{l+1} which maximize $W^{l+1}(T) = \sum_{(i,j)\in T} w_{i,j}^{l+1}$. Transform the spanning tree to an equivalent bifurcating tree T^{t+1} satisfying $l(T_*^{l+1},t^{l+1}) = l(T^{l+1},t^{l+1})$.

SEMPHY repeats the above iterations until convergence to the local maximum. This algorithm can efficiently evaluate all possible trees in each iteration by finding a spanning tree first instead of directly finding a bifurcating tree which maximizes the likelihood. There is a standard algorithm [15] for finding spanning tree. For efficient computation in E-step and M-step I, dynamic programming is incorporated to calculate the expected counts $E[S_{i,j}(a,b)]$ in E-step. Each iteration of the structural EM algorithm is called "Semphy step" throughout the following paper.

2.2. The Best-branch-length (BBL) algorithm

In each iteration of the structural EM algorithm, BBL step in SEMPHY finds the best edge lengths of the tree immediately after finding a step-optimized topology. The EM algorithm of BBL consists of two sub steps which are similar to the E-step and M-step I of the structural EM algorithm except that the calculations are performed on the edges instead of all links of the tree .

3. SEMPHY Parallelization

SEMPHY has to do a number of EM iterations to construct the final phylogenetic tree, which is also time prohibitive especially for the large scale data sets. Parallelization of SEMPHY will make it more powerful to handle large data sets with hundreds of taxa, which covers most size range of current phylogenetic analysis.

Due to the intrinsic EM learning characteristics, we can only perform parallelization within each iteration of EM step, where each EM step will be further decomposed into several kernels. Therefore, application parallelization becomes a problem of how to efficiently parallelize these smaller kernels since the underlying algorithm does not allow higher level concurrency.

To compare the performance on different parallel systems, e.g., SMP (shared memory system) and Clusters, two paradigms, i.e., OpenMP programming model [16] and MPI, are used to parallel SEMPHY with different parallel environments.

3.1. Kernel identification and analysis

The schematic flowchart of SEMPHY is shown in Fig.1, where NJ tree means constructing an initial tree by the Neighbor Joining method. MST denotes finding the spanning tree and transforming it into an equivalent bifurcating tree. There are two major modules in the EM step: Semphy step and BBL step. Other modules such as NJ tree and MST take relatively less time.

When further investigating the Semphy and BBL step, we find that they can be decomposed into several even smaller kernels: calculation of the up and down message (CUD), calculation of counts for edges (CCE), and distance calculation (CD). As depicted in Fig 1, these three kernels are shared by the Semphy and BBL modules. Additionally, Semphy step has a unique kernel, calculation of counts for separate nodes (CCS). To analyze the time distribution of these kernels, the algorithmic complexities for these kernels are summarized as follows: CUD and CCE have $O(|\sum|^2 NM)$ time complexity, and the CCS and CD have $O(\sum|^2 N^2 M)$ time complexity. CCS and CD kernels

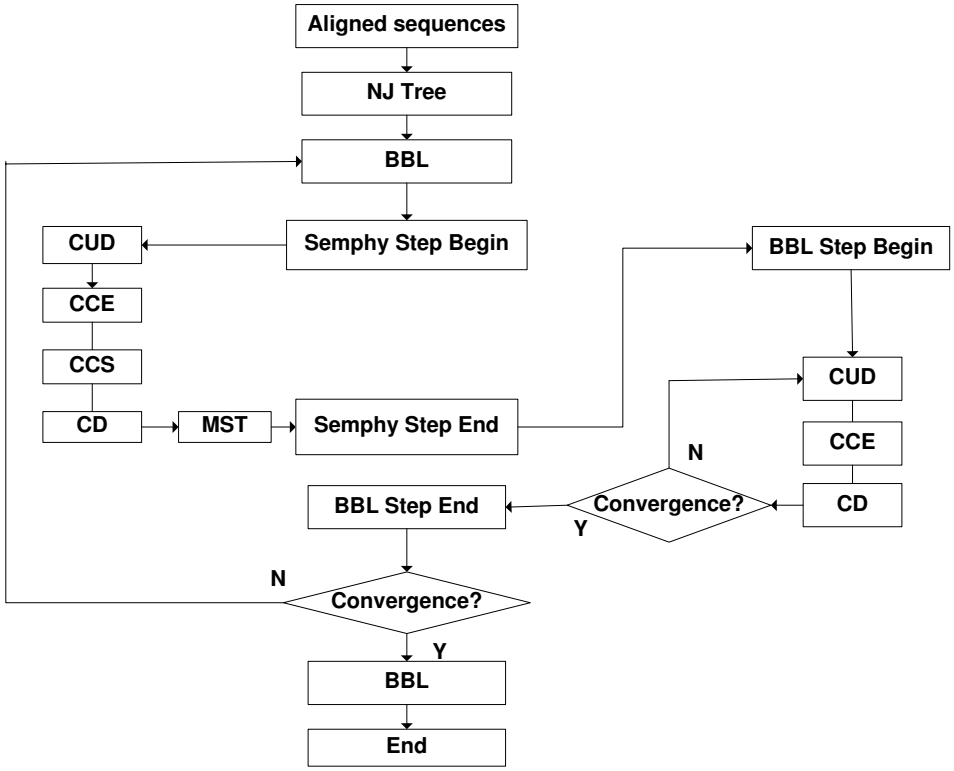


Figure 1. Flowchart of SEMPHY

will become dominating with the increase of the taxa number because the number of separate links increases in quadratic while that of edges increases only linearly. Table 1 shows the quantitative breakdown on three data sets from the Pfam database [17], which confirms the idea that the percentage of CCS and CD kernel keeps increasing with more sequences involved. Furthermore, it can be easily observed that these four kernels are the hotspots from Table 1, which occupies almost 99% of the total execution time. Parallelization can be performed on these time-consuming kernels to enhance the whole application performance.

Table 1
Kernel Breakdown for SEMPHY

Kernels	CUD	CCE	CCS	CD	Others
53 sequences	12.3%	21.88%	20.67%	44.24%	0.91%
108 sequences	8.51%	14.97%	31.0%	45.25%	0.27%
220 sequences	6.14%	9.82%	32.7%	50.6%	0.74%

3.2. Application parallelization

As aforementioned, the majority time are spent on the four kernels (CUD, CCE, CCS and CD). The parallelization of each kernel is specified below.

A. CUD

Table 2
HW/SW environments

Processor speed	3.0G	3.0G
L2 Unified Cache	512KB	512KB
L3 Unified Cache	4MB	/
L4 on-board Unified Cache	32MB	/
Interconnection	Crossbar	Gigabits Ethernet
Memory Size	8GB	1GB
Os	SUSE Linux	Redhat Linux

CUD, which computes up and down messages, including four layer nested loops. We choose the outside loop, the sequence position, to parallelize. The whole iterations are simply partitioned into N segments and they are equally distributed to the N processors. Apparently the scalability of this kernel is limited by the sequence length L and it will incur slight imbalance among all the processors when L is comparable with N.

B. CCE and CD

In SEMPHY, CCE is immediately followed by the CD kernel. Therefore they can be combined together for parallelization. Counts computations for edges are equally assigned to each processor by the edge number, and the subsequent distance computation (CD) also follows the similar decomposition strategy.

C. CCS

The process of CCS is similar to CCE. The difference is that CCE only handles all the edges, whereas CCS handles all the links in the topology tree.

In SEMPHY, the kernels for parallelization often consists of nested ”for” loops, and each item in this loop is independent with each other. Therefore the basic ”parallel for” pragma in OpenMP can be employed to handle most of these cases. Furthermore, we use dynamic scheduling policy to efficiently minimize the load imbalance impact. In spite of all of those general parallel constructs, there are still some special cases, e.g., ”while” loops and C++ iterators, and we choose an alternative Intel omp extension taskq progma to solve it, where the taskq and task pragma uses a work queuing model, serving as the extension of the OpenMP standard by Intel. Since there are few global variables and data sharings in SEMPHY, the OpenMP version can simply dispatch the computations to working threads by adding some compiler directives, without paying much attention to how to manage all of those shared variables.

Compared with OpenMP version, MPI version is considerably more difficult. It has to partition and combine the tasks by hand, carefully deal with all the communications among the working nodes. In practice, the communication cost grows dramatically with the increase of data set due to frequent collective operations, and it becomes a primary scalability limiting factor in the cluster system.

4. Results

To measure the performance of SEMPHY application, two different parallel systems are used: 16-way shared memory system and 16 node cluster. Detailed configurations of these two systems can be found in Table 2.

Table 3
Speedup performance with different dataset

	16 way SMP					
	53.phy		108.phy		220.phy	
	Time(s)	Speedup	Time(s)	Speedup	Time(s)	Speedup
1	271.24	1.0	889.9	1.0	3493.2	1.0
2	136.76	1.98	434.2	2.05	1751.5	1.99
4	71.15	3.81	222.2	4.0	873.4	4.0
8	37.83	7.17	115.3	7.72	436.7	8.0
16	23.04	11.77	62.78	14.17	224.7	15.6
	16 node cluster					
	53.phy		108.phy		220.phy	
	Time(s)	Speedup	Time(s)	Speedup	Time(s)	Speedup
1	201.47	1.0	631.1	1.0	2464.9	1.0
2	105.93	1.90	334.2	1.89	1254.5	1.96
4	58.63	3.44	186.4	3.39	663.6	3.71
8	37.25	5.41	107.9	5.85	376.7	6.54
16	29.13	6.92	72.9	8.66	234.1	10.53

For the test data, we use protein sequence data sets from the Pfam database [17], where three typical sequences: A2M_N.phy (taxa=53, length=394), AA_kinase.phy (taxa=108, length=397), and Aototransporter.phy (taxa=220, length=389) are chosen in our experiments. For simplicity, we use the taxa number to denote these sequences, e.g., 53.phy represents A2M_N protein sequence.

4.1. Scalability performance

Table 3, Fig 2,3 show the basic speedup performance for these data sets respectively. The speedup for the smaller data set, e.g. 53.phy, is linear on 2, 4 and 8 processors, but starts deteriorating when 16 processors is used on the SMP system. The slowdown on more processors occurs because the granularity of the work assigned to each processor decreases. With the increase of data set, we get much better speedup curves.

Comparing these two different parallel paradigms, the OpenMP version is consistently superior to the MPI version in terms of speedup performance. This is not surprising since the MPI version involves tremendous collective operations and the communication overhead increases with more processors.

4.2. Parallel Metrics on SMP system

To discover why SEMPHY exhibits good scalability performance on the SMP system, we further characterize its parallel performance with VTune [18] and Intel VTune Thread Profiler [18]. Table 4 and Table 5 show the metrics in detail.

In Table 4, the SEMPHY workload displays very low barrier, locks and synchronization overhead. The sequential area and the slight load imbalance may hurt the scalability performance on more processors. However, when the problem size increase, we can obtain much better performance. For example, 220.phy demonstrates a consistent better speedup curve than 108.phy on more processors.

Table 5 shows that the memory hierarchy is used efficiently for SEMPHY application, exhibiting very few cache misses, little bus conflicts and low memory bandwidth requirement, which indicates

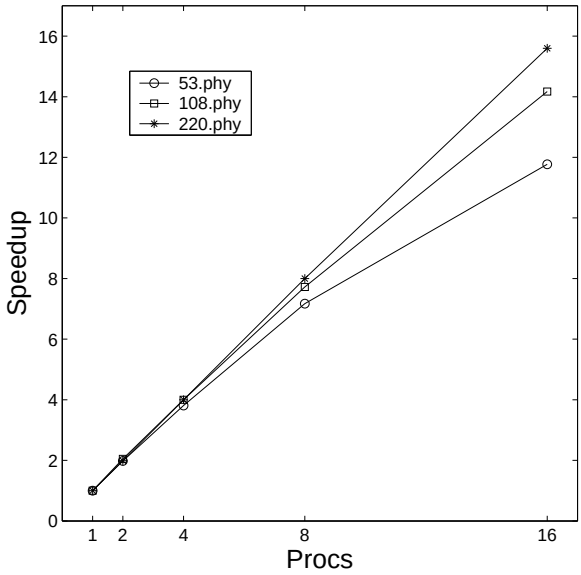


Figure 2. Speedup performance on 16-way SMP (left)

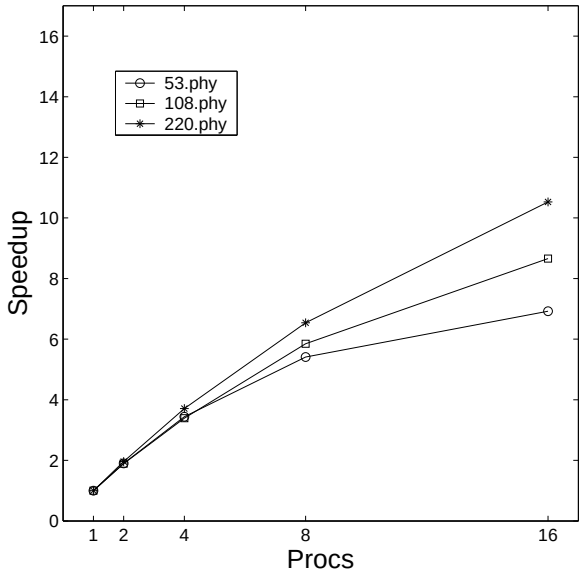


Figure 3. Speedup performance on 16 node cluster (right)

that SEMPHY can scale up well on shared memory systems.

Overall speaking, the SEMPHY application exhibits very good scalability performance in the shared memory multiprocessor system, whereas frequent collective operations and large amount of data communication inhibit its speedup on the cluster environment. It can be expected that the increasing of the hardware resource such as processor numbers on the SMP system will increase its scalability performance accordingly.

5. Conclusions

Maximum likelihood analysis is the most computationally intensive approach to phylogenetic inference. SEMPHY is dramatically faster than other ML approaches while reserving comparable accuracy. In this paper, we present a parallel implementation of SEMPHY, extracting kernels within one iteration of EM step and paralleling them with two different parallel programming paradigms. The experiments demonstrate that SEMPHY scales well on the shared memory system, achieves up to 15.6x speedup on 16 processors. However, the performance on the cluster environment is not satisfying due to huge communication cost.

References

[1] M.Holder, et al: Phylogeny estimation: traditional and Bayesian approaches. Nat. Rev. Genet., 4: 275-284. 2003.

[2] C.A.Stewart, et al: Parallel implementation and performance of fastDNAml: a program for maximum likelihood phylogenetic inference. SC2001. 2001.

[3] J.Felsenstein: The number of evolutionary trees. Syst. Zool., 27:27-33. 1978.

[4] H.L.Bodlaender, et al: Two strikes against perfect phylogeny. In Proceedings 19th International Colloquium on Automata, Languages and Programming, pp. 273-283, Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1992.

Table 4
General parallel metrics with 220.phy sequence

Proc Num	Parallel Area	Sequential Area	Imbalance	Barrier	Locks	Sync.
2	99.7%	0.25%	0.04%	0.0%	0.0%	0.0%
4	99.1%	0.74%	0.17%	0.0%	0.0%	0.0%
8	97.9%	1.6%	0.5%	0.0%	0.0%	0.0%
16	95.2%	3.0%	1.8%	0.0%	0.0%	0.0%

Table 5
Memory hierarchical metrics with 220.phy sequence

Procs	L1 miss rate	L2 miss rate	L3 miss rate	Bandwidth(MB/s)	Bus Access Latency(clks)
1	2.9%	12.4%	8.2%	23.4	183.6
2	3.6%	10.3%	9.0%	50.3	190.2
4	3.5%	12.0%	7.5%	106.4	194.4
8	3.3%	10.7%	9.3%	216.7	188.2
16	3.5%	10.6%	9.3%	447.9	191.7

[5] D.L.Swofford, et al: The phylogenetic handbook, pp. 160-206, Cambridge University Press. 2004.
[6] Ribosomal Database Project-II Release 9.22. http://rdp.cme.msu.edu.
[7] Tree of Life. http://tolweb.org/tree/phylogeny.html.
[8] J. Felsenstein: Inferring Phylogenies, Sinauer Associates, Sunderland, Massachusetts. 2003.
[9] J.J.Wiens, et al: Phylogenetic analysis and intraspecific variation: performance of parsimony, likelihood, and distance methods. Syst. Biol., 47: 228-253.1998.
[10] G.J.Olsen,et al: fastDNAml: A tool for construction of phylogenetictrees of DNA sequences using maximum likelihood. Comput. Appl. Biosci. 10: 41-48. 1994.
[11] N.Friedman, et al: A structural EM algorithm for phylogentic inference. J. Comput. Biol., 9: 331-353. 2002.
[12] N. Friedman: A Structural EM algorithm for Phylogenetic Inference. Journal of Computational Biology, 9:331-353, 2002.
[13] N. Friedman. Learning belief networks in the presence of missing values and hidden variables. In Fisher, D. ed., Proceedings of the Fourteenth International Conference on Machine Learning, pp. 125-133, Morgan Kaufman, San Francisco, 1997
[14] J. Adachi, et al: Molphy version 2.3, programs for molecular phylogenetics based on maximum likelihood, Technical report, The Institute of Statistical Mathematics, Tokyo, Japan.1996.
[15] Kruskal, J.B.: On the shortest spanning subtree of a graph and the traveling salesman problem, Proceedings of the American Mathematical Society, 7: 48-50, 1956.
[16] OpenMP Architecture Review Board: "OpenMP C and C++ Application Program Interface," Version 2.0, March 2002, http://www.openmp.org.
[17] Bateman, A., et al: The Pfam Protein Families Database. Nucleic Acids Research Database Issue 32:D138-D141, 2004.
[18] Intel Corp.: Intel Vtune Performance Analyzer, (available on-line: http://developer.intel.com/software/products/vtune/).

Exploiting parallelism on irregular applications using the GPU *

Manuel Ujaldon[a], Joel Saltz[b]

[a]Computer Architecture Department. University of Malaga. Complejo Tecnologico. Campus Teatinos. Malaga, 29071. SPAIN

[b]Biomedical Informatics Department. Ohio State University. 3197 Graves Hall. 333 W. 10th Ave. Columbus, Ohio 43210. U.S.A.

The computational speed on microprocessors is increasing faster than the communication speed, especially on parallel processors such as GPUs. Thus, the computations that benefit the most from GPU processing have high arithmetic intensity. This paper compares the effectiveness of GPUs when handling scientific general-purpose irregular problems, outperforming counterpart CPUs by a wide margin and identifying the AGP bus as the major bottleneck in graphics architecture. We study the impact that the emerging PCI-Express bus has for accelerating such applications when replacing AGP. A number of software optimizations are also conducted by using recent APIs, OpenGL extensions and drivers, leading to loading times 40% lower on PCI-Express and four times faster when overlapping communication with computation. Execution times are shown on a benchmark composed of an Euler solver and a sparse matrix-vector product running on Nvidia GeForce FX and GeForce 6800 GT graphics cards.

1. Introduction

By taking advantage of the streaming processing model, modern graphics processors (GPUs) are outperforming their CPU counterparts in some general-purpose applications, and the difference is expected to grow in the future [7].

Modern CPUs have been increasing their performance according to Moore's Law over the last three decades. Such improvements have been mostly based on clock frequency and transistors manufacturing process, which find severe boundaries for progressing in the future. GPUs, on the contrary, double performance every six months relying on memory latency rather than on raw speed. Their improvements are focused on architectural layers, by setting a streaming execution model which reverses the bottleneck inherent to memory access: Data are the axis flowing through the graphics pipeline, and instructions are those who come to meet them. Since there is no memory hierarchy nor data dependencies in the streaming model, the pipeline maximizes throughput without being stalled. That way, whenever the GPU is consistently fed by input data, performance boosts, leading to an extraordinary scalable architecture.

We have taken advantage of these extraordinary capabilities by developing methods for mapping irregular general-purpose algorithms onto the GPU [12,13], where we outperform CPU performance by a 2x-4x factor in execution time. Nonetheless, a major bottleneck located in the AGP bus affected performance when accounting the time for loading the input data onto the GPU.

This paper contributes to overcome such bottleneck by exploring the features of the PCI-Express bus recently introduced for the graphics cards in commodity PCs, and performing a number of optimizations using OpenGL extensions. Other issues regarding data communication and accessing

*This work was partially supported by the Ministry of Education of Spain, through the Secretary of State for Education and Universities, grant PR2004-0508.

Graphics processing	Conventional programming	Graphics processing	Conventional programming
Texture memory	Arrays in main memory	Geometry (T & L)	N-ary arithmetic operators
List of vertices	Inner loop(s) of a code block	Blending functions	Reduction operators
Rendering passes	Outer loop of a code block	Clipping the scene	IF within the inner loop
Vertex indexing	First (inner) level of indirection	Active window	IF within intermediate loops
Textures lookup	Intermediate levels of indirection	Color index mask	IF within the outer loop
Color tables	Last (outer) level of indirection	Multipass rendering	Kernel programming

Table 1
The GPU abstraction basics for a programmer.

are also investigated in our work, namely: (1) The memory allocation scheme for the input data, providing hints to OpenGL for an optimal data placement within the graphics card. (2) The new representation for color channels using 16-bit floating-point numbers (as introduced by NVIDIA in the GeForce 6 series), which allowed us to improve the accuracy in our results with little cost in execution time. (3) The driver impact on recent hardware developments, particularly PCI-Express.

The rest of this paper is organized as follows: Section 2 briefly introduces our methods for implementing general-purpose irregular algorithms within the GPU. Section 3 introduces the irregular kernels we use as benchmark. Section 4 describes alternatives for programming GPUs and discusses its influence in functionality and performance. Section 5 shows execution numbers for our benchmark compared with those of the CPU. Section 6 introduces our optimizations for reducing the graphics bus congestion. Section 7 summarizes related work, and finally Section 8 draws the conclusion from our work.

2. Our approach for mapping irregular computation onto the GPU

A typical graphics processor accepts an input stream (vertex attributes), transform it through a sequence of kernels or *shaders* (vertex program, fragment program, texture operators), and return an output stream (rasterized pixels), which is written into the frame buffer. Using GPUs for general-purpose computation entails disguising input data as vertex attributes, large data structures as textures, instructions as kernels, and final results as portions of video memory.

All these elements can be accessed by programmers using APIs such as DirectX or OpenGL. They just have to forget the traditional programming paradigm and focus on the data flow (the stream). Basically, each building block of a program constitutes a stream of vertices, whose geometry is defined according to existing loops and conditionals in the block for the kernels to compute only the desired elements. Multipass rendering executes the blocks sequentially, with the frame buffer and textures memory being used for the communication between consecutive blocks.

Table 1 shows a list of GPU-CPU equivalencies extracted from our experience when implementing codes on the graphics processor. More details on how to exploit data locality, map operators and implement indirect array accessing on the GPU can be found in [12,13]. Overall, GPUs are used for different purposes they are intended to, and our goal is to identify those valuable resources for irregular computation which can lead to a performance gain on the GPU. As it can be observed on the table, multiple level of indirections when accessing indexed arrays can be solved directly on the GPU hardware using vertices, textures and colors. In addition, we propose to implement reduction operators as blending functions to enhance performance versus the CPU.

Nonetheless, the GPU might well be seen more as a cooperator than a rival to the CPU, using executeAsync() calls to exploit task parallelism on a coarse grain algorithm decomposition: Since we deal with small kernels here, they might as well be considered as potential tasks the CPU delegates to the GPU as parts of a larger application.

Data sets size	Small	Medium	Large
Euler nodes	2800	9428	53961
Euler edges	17367	59863	353476
File Size (Kb)	576	1810	11524

Matrix Rows	3948	13992	28924
Matrix nonzeros	60882	316740	1036208
Matrix Fill rate	0.39 %	0.16 %	0.12 %
Matrix File Size	1568 Kb	2680 Kb	8628 Kb

Table 2
The input data set used for our benchmark. Left: Euler solver. Right: SpMxV.

3. The benchmark

We execute a couple of typical irregular kernels dealing with indirect array accessing:

1. The **Sparse Matrix-Vector Multiply (SpMxV)**, used as kernel in iterative methods within linear algebra. Indirections here are a consequence of the data structures used for storing matrix nonzeros in compressed formats (see Figure 1). The input vector, X, is stored as a texture, whereas Column is used as texture coordinates for accessing X, Data is stored as the color attribute for vertices, and Row lies in vertex positions defined to merge results onto the frame buffer holding Y.

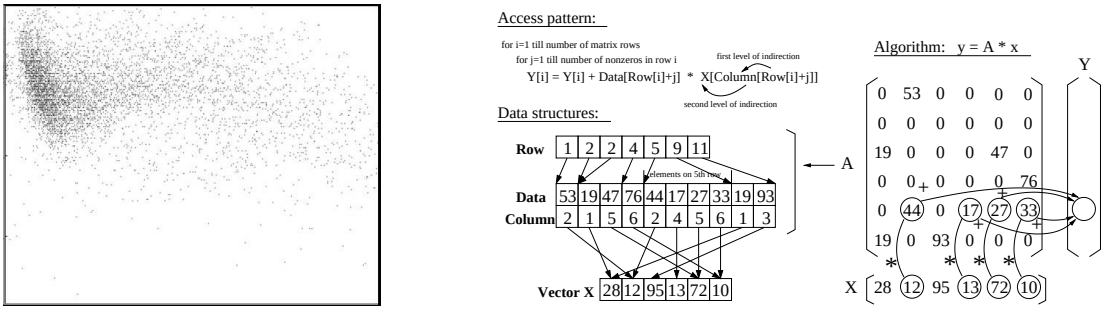


Figure 1. Data structures and access pattern for the sparse matrix-vector multiply (SpMxV).

2. **The Euler solver**, an adaptive partial different equation solver sweeping over unstructured meshes for discretizing complex domains and calculating forces between all pair of nodes connected through defined edges (see Figure 2). Major differences with respect to the SpMxV are the presence of indirections on the left hand side of assignments and the compact way for computing 3 statements in a vector manner using the RGB color space for the 3 components in the 3D force.

We decided in favour of iterative algorithms in order to perform a survey about the loading time into the GPU for the input data versus the number of iterations the GPU has to perform to amortize this cost. The AGP bus was already revealed as a potential bottleneck in former experiments, and so we selected applications where the computation/communication ratio might be high but at the same time variable, with the aim of testing its influence in GPU performance.

With a similar purpose, we selected input data sets of small, medium and large sizes for running the experiments (see Table 2). The left side of the table summarizes the features in three unstructured meshes used for the Euler solver, where nodes are connected through edges with an average connectivity of six. The input data set was taken from real applications at ICASE NASA Lab. The right side contains the parameters defining three different sparse matrices taken from the Harwell-Boeing collection, where they are represented in compressed row storage format.

4. Programming the GPU

4.1. Memory allocation

Using Vertex Buffer Objects (VBO) in OpenGL we were able to process vertices in custom ways without having to shuffle them between main memory and the card, something particularly useful

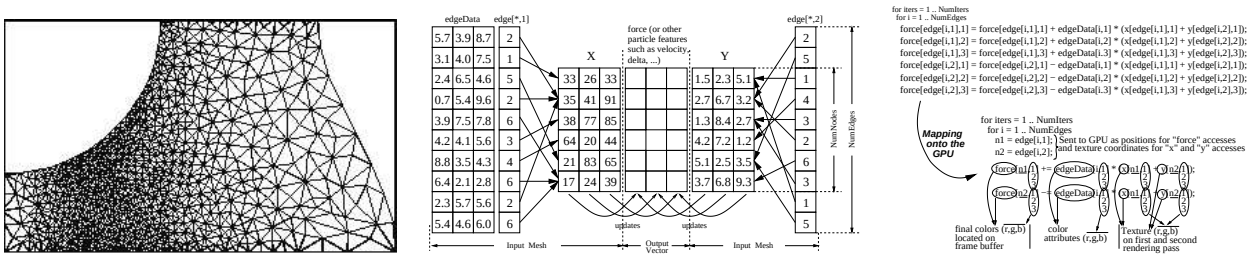


Figure 2. Data structures and access pattern for a 3D Euler kernel sweeping over an unstructured mesh.

Year	CPU	Main Memory	GPU	Video Memory
2003	Pentium 4 @ 2.4 GHz	1 Gb @ 4.2 Gb/s	GeF FX 5900 @ 450 MHz	128 Mb @ 27.2 Gb/s
2004	Athlon 64 @ 2.0 GHz	2 Gb @ 6.3 Gb/s	GeF FX5950U @ 475 MHz	256 Mb @ 30.4 Gb/s
2005	Pentium 4 @ 3.2 GHz	1 Gb @ 8.4 Gb/s	GeF 6800 GT @ 350 MHz	256 Mb @ 35.2 Gb/s

Table 3
Hardware features for our CPU-GPU comparison.

when a large amount of repeating geometry is involved in the GPU computation. This was always the case in our iterative algorithms, where access pattern (which defines the problem geometry for the GPU) remains unchanged through iterations. For further optimizations or when the buffer becomes too big for the memory available, we build block schemes switching between smaller portions of the buffer. This was carried out with the EXT_compiled_vertex_arrays OpenGL extension, which allowed us to lock and unlock vertex arrays for caching.

We started applying these schemes to the SpMxV code by strip-mining the loop sweeping over rows, and switching the Column vector between rows while keeping the X vector always within the VBO (its double indirection makes it to be the most unpredictable pattern). With all these optimizations, execution times for the SpMxV were 3-4 times faster, so we extended them to the Euler kernel as well, where the gains achieved were more modest. The ARB_Vertex_Buffer_Object extension was included in OpenGL 1.5, and later extended to pixels with the ARB_Pixel_Buffer_Object extension (December, 2004), an additional possibility we haven't tested in our experiments yet.

4.2. Floating-point precision

We tested the impact of computing reduction operators using the final blending stage in the graphics pipeline. In particular, the SpMxV algorithm performed the accumulation process of partial products this way (see right side on Figure 1), by reserving an area in the frame buffer for the result vector Y. Till the arrival of the GeForce 6 series, all these operations were poorly implemented on GPUs by using just 8-bit floating-point numbers associated to color representation in the RGB space.

This was a source of inaccuracy for the GPU within scientific computing, and remained to be seen whether computing on higher precision was going to hurt performance quite a bit. We measured the penalty for performing such operations using higher precision on a GeForce 6800 GT card versus an older GeForce 5950 Ultra model, and the execution times remained almost unaffected (roughly, those numbers correspond to the last two bars on each of the charts shown in Figure 3). We conclude that GPU is suffering from floating-point inaccuracy nowadays, but further developments towards 32-bit floating-point arithmetic will affect positively the precision without hurting performance severely, which is a good insight when thinking of GPUs as future general-purpose processors.

5. GPU performance versus CPU

Our OpenGL code with VBO and 16-bit floating-point precision in blending functions was compared against the CPU on regular PCs equipped with the latest CPUs and NVIDIA graphics cards from the GeForce 5 and 6 series.

On the CPU, we use Visual C++ 7.0 running under Windows XP. Multimedia extensions (SSE on Pentium 4 and 3DNow! Professional on Athlon 64) were enabled relying directly on HAL layer without any specific library in between. Cache performance was optimized sweeping consecutive memory addresses through loop reordering. Single-precision numbers were used as floating-point.

On the GPU, OpenGL 1.4 and 1.5 was used, plus a number of OpenGL extensions for subsequent optimizations: (1) **ARB_vertex_buffer_object** (Feb'03) allowed us an efficient memory allocation for vertices and colors onto the GPU (as discussed in Section 4.1). (2) **NV_texture_shader** (May'04) made it possible to tune the internal GPU texture shaders to our particular needs. (3) **NV_vertex_array_range** (Sep'01) and **NV_fence** (November 2003 - see [11]) allowed us to overlap CPU-GPU communications with GPU computation at different levels and switch vertices allocation between video and AGP memory. (4) **EXT_pbuffer** (Jan'99) enabled writing the results in areas different than the frame buffer, and reuse them in subsequent rendering passes. (5) **NV_float_buffer** (Jan'03) added floating-point support for textures, selecting their particular format as well as its association with color channels.

In addition, we used OpenGL interleaved arrays for sending vertex attributes to the GPU, vertex positions were precisely calculated according to screen resolution to skip interpolations, and GL_POINTS was selected as drawing primitive to keep computations strictly on the input list of vertices. Times in our benchmark were measured with the QueryPerformanceFrequency() and QueryPerformanceCounter() functions available in Visual C++. The nView tuning tool from NVIDIA was used to disable antialiasing, dithering and anisotropic filtering, mip filter and sample optimizations. Hardware acceleration was set to Single Display Mode and image setting was set to High Performance. Vertical synchronization was disabled since it limits the frame rate to the refresh rate of the particular monitor attached to the PC.

5.1. Execution time

Execution times for the CPUs and GPUs are shown in Figure 3, with the upper row corresponding to the Euler kernel and the lower row to the SpMxV code. On each chart, the three bars on the left are CPU times; the ones on the right belong to GPUs. In order to perform a fair comparison, first and fourth bars correspond to the same PC, and so do second and fifth as well as third and sixth.

It can be seen that performance is between a 2-4 factor in favour of GPUs, and that the difference is increasing with the size of the data set. However, the burden for loading the input vertices and textures increases with the data set as well (see Figure 4.a)

5.2. Communication time

PCI-Express replaces the parallel multidrop architecture of AGP with a serial, point-to-point connection bus [2]. Current bandwidth reaches 4 Gbytes/s., doubling AGP 3.0 plus an extra factor of two since PCI-Express is a dual-simplex specification willing to communicate both ways simultaneously. Besides, PCI-Express frequency can still be increased up to 10 gigatransfers per second from the current value of 2.5, and bus width can be doubled to 32x from the actual 16x implementation.

Figure 4.a illustrates PCI-express performance when used for loading data in our benchmarks. As compared to AGP, the overhead was reduced by around 40% on the larger data set. Our results show that AGP loading times represent roughly 20 times the execution time, whereas on PCI-Express they

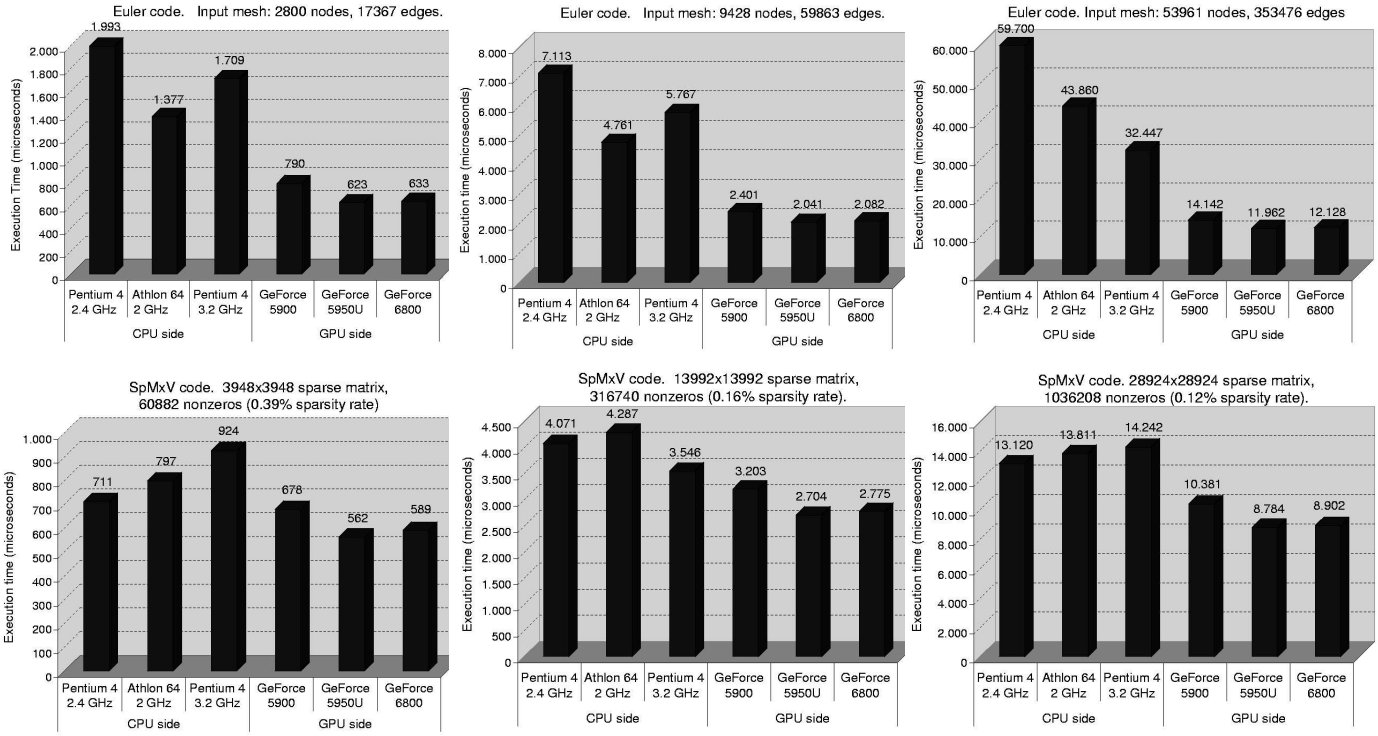


Figure 3. CPU-GPU comparison in execution time for the Euler and SpMxV codes (excluding loading times).

come down to nearly 12 times. Moreover, without PCI-Express the Euler code has to execute seven iterations for the GPU to defeat the CPU on the larger execution when accounting the communication time. Using PCI-Express, this requirement is cut to only four iterations. For the SpMxV kernel, the number of iterations required to amortize this cost are 85 for AGP, and 55 for PCI-Express.

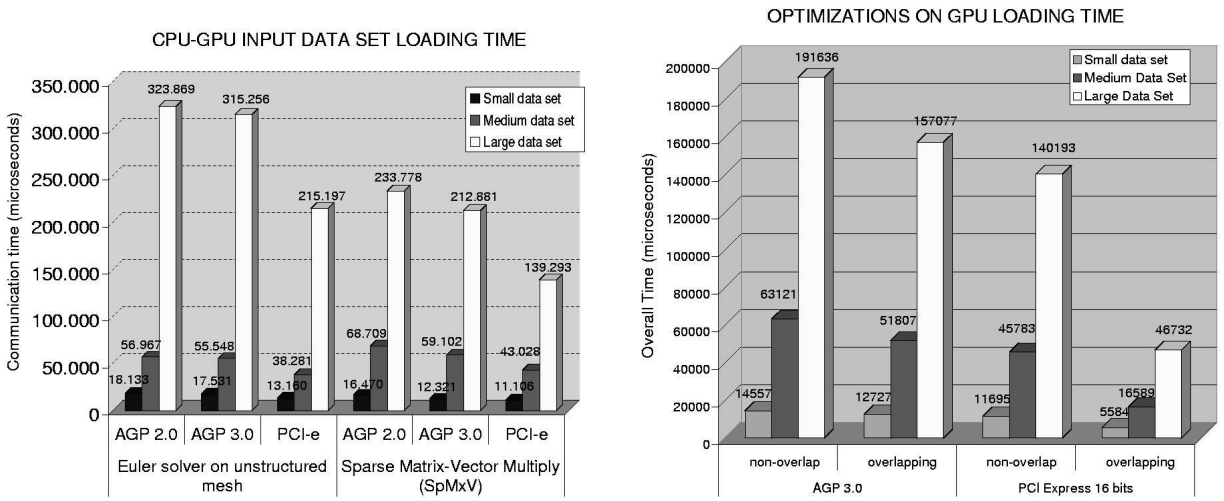
The huge difference between the two codes lies in the compact manner that Euler describes the geometry: A single (x,y,z) position and (s,r) texture coordinate is shared among the 3 components of the Force, X and Y arrays. This is opposed to the SpMxV code, where the size of the communication buffer is 8 times the number of nonzeros (for each vertex, we have (s,r), (r,g,b) and (x,y,z)).

6. Overlapping communication/computation

Computation time comprises the actual time that data is processed on the GPU; loading time includes the tasks for converting the input data sets into graphical data structures and passing them onto the GPU. To speed-up the entire process while decoupling the GPU computation from its communication needs, we use the NVIDIA OpenGL extension NV_vertex_array_range to place vertices directly onto GPU accessible memory. Then, we overlap communication and computation using the NV_fence extension, a fine grained synchronization mechanism available in OpenGL [11].

We allocate a circular buffer in video memory and partition it into several smaller buffers. The CPU now places a part of the vertex data into the first partition, gives the call to the GPU to process this data, and while the GPU pulls and computes the data, the CPU places another part of the vertex data for the next partition and so on.

Figure 4.b shows the results we obtained when performing this optimization over the SpMxV code. Due to the overlapping, we cannot split loading and execution time, nor decouple one iteration from another. Instead, we performed 100 SpMxV iterations and then accounted for all the commu-



(a) (b)

Figure 4. (a) AGP/PCI-Express comparison for loading the small, medium and large meshes (Euler) and sparse matrices (SpMxV). (b) Overall execution time (loading times plus a single rendering pass) for the SpMxV when overlapping CPU communications with GPU communications on different matrix sizes.

nication time that was not overlapped with computation, taking that as the actual communication overhead. Loading times were this way greatly reduced, roughly 20% for the AGP bus and almost 70% on PCI-Express, where snooping was revealed as an outstanding mechanish for maximizing the desired overlap. We also measured the effect of having more recent drivers in the PCI-Express machine than in the AGP PC, finding out the drivers to be responsible for just marginal gains.

7. Related Work

The use of graphics hardware for executing general-purpose applications is becoming increasingly popular ([1]). Some examples outperforming CPUs are volume segmentation (10-20 times faster) [10], surfaces deformation (10-15x) [9], multigrid solvers (3x) [6] and linear algebra (2x) [3,8]. In all those cases, the operations are expressed in terms of appropriate graphics operators, though the output was first constrained to integers and later to low precision floating-point data.

Currently, 32-bit precision is supported on several graphics cards, but some constraints still remain at the API level: In OpenGL, stages like rasterization clamp vertex attributes to the (0..1) range unless you use programmable shaders to bypass those operations, and blending functions or texture handling can process only 16-bit floating-point numbers, which is the most sofisticated color representation up to date. Besides, DirectX doesn't allow you to specify data 32-bit long in certain operations. The results we show through this paper are also affected by such limitations.

Data reuse and building computational blocks on the GPU was also an aspect recently investigated by Fatahalian et al. [5], who implemented a dense matrix-matrix multiplication on the GPU using programmable shaders to conclude that the lack of cache memory will limit GPU performance. We don't find cache that valuable when the access pattern is irregular, and besides we prefer to avoid the use of shaders because that would force us to decompose the problem using multirendering.

Even though shaders restrictions may be relaxed in the future, it means computing the access indices at run-time, a major burden when indirections predominate. In cases like the SpMxV, where the access pattern remains constant through iterations, we set up the geometry for vertex position to act as a tag for guiding the streaming computation, which qualify us for extracting the entire index calculation out of the execution loops and amortizing the loading time through iterations.

8. Conclusions

In this paper, we implement a couple of irregular computational kernels onto the graphics pipeline, showing how general-purpose applications can benefit from a streaming execution model to outperform current CPUs by a wide margin . Our methods avoid programming the shaders to overcome their current limitations, and benefit from a bunch of OpenGL extensions to compute the whole algorithm on a single rendering pass.

We focus this work on the experimental side at different levels of the graphics card: (1) API: OpenGL turned out to be faster than DirectX, and also offered us much richer extensions for further optimizations. (2) Memory allocation: Vertex Buffer Objects became an efficient mechanism for reusing data geometry through iterations and building computational blocks. (3) Floating-point precision was not hurting performance when enhanced from 8-bit to 16-bit in the final stages of the graphics pipeline, as already available in the GeForce 6 Series by Nvidia. (4) Communication time accounted for most of the running time and was vastly reduced, first 40% using PCI-Express and then up to an additional 70% when overlapping with computational times through a large number of iterations. Further improvements will be reached when PCI-Express shows its extraordinary scalability in the future, removing the actual bottleneck from the graphics card.

Driven by the game industry, future graphics cards are expected to continue increasing their capabilities so that current restrictions on shaders, floating-point accuracy and communication overhead will be relaxed and virtually any application can be mapped onto the graphics pipeline.

References

[1] A Web page dedicated to the latest developments in general-purpose on the GPU. http://www.gpgpu.org.

[2] A.V. Bhatt. *Creating a Third Generation I/O Interconnect.* Intel Developer Network for PCI Express.

[3] Bolz, J., Farmer, I., Grinspun, E., Schroder, P. *Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid*. Procs. (SIGGRAPH'03). San Diego (California). July, 2003.

[4] Duff I.S., Grimes R.G., and Lewis, J.G. *User's guide for the Harwell–Boeing sparse matrix collection (Release I).* Technical Report TR/PA/92/86, CERFACS, Toulouse, 1992.

[5] K. Fatahalian, J. Sugerman, and P. Hanrahan. *Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication.* Procs. (HWWS'04). Grenoble (France), August, 2004.

[6] Goodnight, N., Woolley, C., Lewin, G., Luebke, D., and Humphreys, G. *A multigrid solver for boundary value problems using programmable graphics hardware*. Procs. (HWWS'03). pp.102-111, July, 2003.

[7] Khailany, B., Dally, W., Rixner, S., Kapasi, U., Owens, J. and Towles, B. *Exploring the VLSI Scalability of Stream Processors*. Procs. 9th Symposium on High Performance Computer Architecture. Anaheim (California), Feb. 2003, pp. 153-164.

[8] Kruger, J., Westermann, R. *Linear Algebra Operators for GPU Implementation of Numerical Algorithms*. Procs. (SIGGRAPH'03). San Diego (California). July, 2003.

[9] Lefohn, A., Kniss, J., Hansen, C., and Whitaker, R. *Interactive Deformation and Visualization of Level Set Surfaces Using Graphics Hardware*. Procs. 14th IEEE Visualization Conference, Seattle (Washington), October, 2003, pp. 75-82.

[10] Sherbondy, A., Houston, M., Napel, S. *Fast Volume Segmentation With Simultaneous Visualization Using Programmable Graphics Hardware*. Procs. 14th IEEE Visualization Conf., Seattle, October, 2003.

[11] Spitzer, J. and Everitt, C. GL_NV_vertex_array_range and GL_NV_fence on GeForce Products and beyond. NVIDIA Corporation. August, 2001.

[12] M. Ujaldón, J. Saltz. *Mapping Irregular Computation onto the Graphics Pipeline.* Internal Report 2004, Biomedical Informatics Dept, Ohio State University.

[13] M. Ujaldón, J. Saltz. *The GPU as an indirection engine for a fast information retrieval.* Intl J. Electronic Business, Ed. Inderscience, vol. 3, number 3/4, pages 316-327. July-August, 2005.

Biomedical and Civil Engineering Experiences Using Grid Computing Technologies

J.M. Alonso[a], V. Hernandez[a], G. Molto[a]

[a]Departamento de Sistemas Informaticos y Computacion, Universidad Politecnica de Valencia, Camino de Vera s/n, 46022 Valencia, Spain

1. Introduction

In the mid 90s, the Grid concept appeared as a new trend in distributed computing. Grid Computing technology [6] emerged as the solution for some of the computational problems of organisations, enabling the collaborative use of remote resources to execute computationally expensive tasks across the world. The main aim of a computational Grid is to provide a coherent view of distant heterogeneous machines so that they act as a single, huge, powerful and self-managed computer. This fact has caused a tremendous impact for computationally intensive applications, which can bene t from the power that a Grid infrastructure aims at delivering.

Nowadays, important research efforts are being dedicated to the development of the software infrastructures that enable to achieve this vision. However, Grid Computing is still in the early stages and the users face a lot of complexity as well as a steep learning curve when applying this technology to their own applications.

In this paper we describe the usage of Grid Computing technologies in two applications belonging to different scienti c elds. The rst one involves the 3D dynamic analysis of large-scale buildings. The second one performs electrical simulations of cardiac tissues. In both applications, an iterative simulation procedure gives place to a computationally and memory-intensive process. Two previously implemented applications have been ported to a Grid infrastructure by developing and using later, a generic software layer called GMarte, which simpli es the process of executing a scienti c software on a Grid-based distributed infrastructure.

To test the bene ts of Grid Computing in the applications considered, executions have been performed both on a local and a large-scale Grid infrastructure. On the one hand, the local infrastructure represents a Globus-based Grid composed of machines from our research group. On the other hand, we have also employed part of the resources available in the framework of the EGEE project, the largest distributed deployment available for e-science.

The remainder of the paper is structured as follows: Section 2 presents the two target applications considered. Then, section 3 details the GMarte software layer employed to develop the Grid applications that enable the target applications to be executed on a Grid. Next, section 4 describes the structural case study that has been executed, detailing the computational infrastructure employed and the results. Later, section 5 reports the execution of a cardiac case study on the large Grid deployment. A discussion of the principal problems that arise when moving from a local to a global Grid is performed in section 6. Finally, section 7 summarises the main achievements and concludes the paper.

2. Target Applications

This chapter brie y describes the two applications chosen for their execution on a Grid. They have been selected because of their large computational and memory requirements.

2.1. Structural Dynamic Analysis of Buildings

3D dynamic analysis of large-scale buildings has been considered by engineers as a challenging problem, owing to its high computational demand. Most of the existing commercial codes are composed of questionable simplifications, reducing the number nodes to be considered and the number of degrees of freedom associated, because the computational and memory requirements involved in a realistic simulation may be too intensive for a traditional computer. Notwithstanding, these simplifications, although appropriate for single structures, have demonstrated to be completely inadequate for complex buildings. Previous research has shown that results from seismic analysis are model dependent and it indicates the need for an adequate and realistic 3D analytical model [11].

In the preliminary stage of a building, structural engineers usually work with different initial designs. Each design can be a different layout, composed of distinct materials (concrete, steel, etc.), where different dimensions are assigned to the members, or even where several external loads, which could occur during the lifetime of the structure, are applied. The large number of resulting structural solutions must be analysed rapidly.

In some cases, all these designs are rejected, thus returning to the initial design stage. Nevertheless, a selection among these solutions has to be made before proceeding to the detailed design phase. Now, an iterative trial-error process takes places by the structural engineer to achieve the most efficient solution where, varying the member dimensions, the whole structure is analysed and the results are interpreted. Obviously, calculations must be performed accurately, complying criteria of safety, cost limitations and construction constraints. Moreover, this number of simulations required is notably increased when dealing with dynamic analysis owing to the need to work with several dynamic loads. As an example, the Spanish Earthquake-Resistant Construction Standards (NCSE-02) requires that a building is analysed with at least five different representative earthquakes. This gives place to a large amount of different structural alternatives to be computed (a structural case study), where a huge computational power is demanded.

In a previous work, an application was developed to perform 3D dynamic analysis of buildings, where all the nodes of the structure are considered and 6 degrees of freedom per node are taken into account [3]. Simulations are carried out by means of 8 well-known direct time integration methods [8].

2.2. Simulation of the Electrical Activity in Cardiac Tissues

According to the European Heart Network[1], cardiovascular disease causes nearly half of all deaths in Europe. In fact, cardiac arrhythmias are one of the first causes of mortality in developed countries. However, despite the intense research in this area, the generation of ventricular tachycardias and ventricular fibrillation (the most mortal arrhythmias) are not clearly understood.

Simulation is an excellent tool that enables to formulate multiple hypotheses *in silico* prior to clinical experimentation. However, the simulation of the electrical propagation in cardiac tissues (a key indicator of the state of the heart) represents a computational and data intensive problem, where the electrical study of a medium-sized tissue can last for several days on a sequential platform. This is mainly due to the usage of comprehensive ionic models which detail the internal behaviour of each cardiac cell being simulated.

A MPI-based parallel simulator was developed for the simulation of the action potential propagation in cardiac tissues. It enabled a substantial reduction in the execution time of a single simulation on a cluster of PCs [1]. However, in addition to this computational cost, there are many research studies that involve the execution of parametric simulations. For example, to test the effects of new

[1] http://www.ehnheart.org

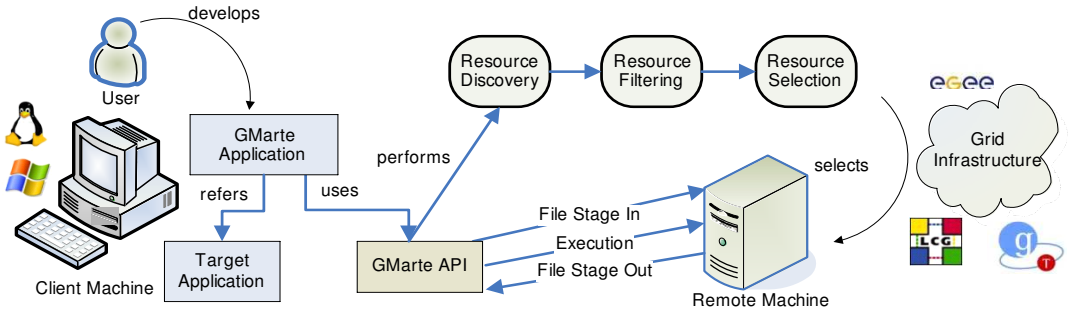


Figure 1. Principal diagram of usage of GMarte.

medicines, it is required to perform multiple simulations varying the drug concentration. As another example, to study the effects of late ischemia it is necessary to vary the junctional resistances in a given interval and investigate the evolution of the electrical activity.

These cardiac case studies, composed of multiple independent parametric executions, enlarge the total computing cost by the number of simulations to be run, what turns the global study procedure into an even larger computational problem.

3. The GMarte Framework

These two different scienti c elds face common problems. They involve the execution of multiple independent parametric simulations. Given that a Grid infrastructure aims at offering a large computing power, a Grid-based solution, that enables to perform multiple concurrent simulations on these machines, would be an ideal solution to accelerate the execution of these problems, which will be denoted as *case studies*.

To support our executions, we have focused on the Globus Toolkit 2.4.3 [7], the current *de facto* standard middleware for computational Grids. This middleware provides protocols and services for connecting resources and deploying large-scale computational global Grids. However, its inherent complexity often discourages scientists from porting their applications to a Grid infrastructure. To overcome this handicap, GMarte (Grid Middleware to Abstract Remote Task Execution) [5] was developed. It consists of an easy-to-use middleware, developed on top of the Java CoG 1.2 [10], exposing an Object-Oriented view of the Grid that enables fault-tolerant metascheduling in a Globus-based framework. GMarte has also been extended to support executions on the Computing Elements of the LCG testbed, which is the largest Grid deployment available for scienti c computation. A more detailed description of this infrastructure will be provided later.

Using the high level API provided by GMarte, the user no longer interacts with the low-level services provided by Globus. Therefore, instead of detailing *how* to perform the executions, the user declares *what* to execute by using the high-level object abstractions offered by the Java API provided (i.e. GridTask, GridResource, etc). An XML approach for the de nition of the case studies is currently under development to enable the case study de nition without depending on the Java API. It is important noting that GMarte is a client-side middleware, only installed on the client machine, which will interact with the computational resources of the Grid infrastructure. In addition, as it is developed in Java, the client is functional on different operating systems, such Windows and Linux. Being out of the scope of this paper, further information about GMarte can be found elsewhere [4].

Figure 1 describes the corresponding usage diagram for a user that wants to execute a particular case study on a Grid deployment. First of all, the user writes a small *GMarte Application* which

describes the tasks to be executed (i.e. instances of the *Target Application*) along with their computing requirements in terms of processors, needed RAM, etc. This application also speci es the set of machines to be employed for execution, either by an explicit enumeration or by delegating into a resource discovery component. Finally, this application may also specify a certain metascheduler (the component that performs the task allocation) from those already implemented in GMarte. Once the case study simulation is launched, the GMarte middleware performs a sequence of steps in order to achieve successful execution of the tasks. First of all, if required, the *Resource Discovery* phase obtains a list of potential machines candidates for execution. This is obtained by querying standard Index Information Services for Globus or LCG. Then, *Resource Filtering* discards those resources that are unavailable for execution, that can not be accessed with the user credentials supplied or that do not ful ll the minimum application-dependent execution requirements. These two phases are only executed once in the scheduling procedure.

Then, for each task, the following phases are performed to achieve remote task execution: First of all, *Resource Selection* involves choosing the current best resource for the execution of the task. The implemented policy in GMarte considers the application requirements as well as an estimate of the data transfer cost to each resource. Besides, a workload component prevents from the allocation of all the tasks to a single resource, what would be critical if the resource fails. This policy can easily be altered by the user in order to guide the scheduling procedure. Once the machine has been selected, the *File Stage In* phase transfers the *Target Application* as well as all its dependent input les from the *Client Machine* to the *Remote Machine*. Next, the *Execution* phase starts the application in the remote machine. Finally, when the execution nishes, the *File Stage Out* phase retrieves the selected output data les back to the local machine, erasing the remote data les.

4. Structural Case Study

To assess the effectiveness of Grid Computing technologies in the 3D structural analysis of buildings, a hotel was simulated. These singular buildings must be carefully designed, because an appropriate solution can be the differential factor for economical pro tability in countries where the tourism is one of the biggest industry. The design of hotel facilities and security requirements imply to consider distinct relevant factors, leading to different structural solutions for the same problem.

In our case, the hotel geometric model was composed of about 100,000 degrees of freedom and two alternatives were considered for the construction material: reinforced concrete and steel-concrete composite frame. For each material, ve combinations of different structural member dimensions were assigned. Finally, and taking into account the NCSE-02, six representative earthquakes were applied to each structural solution. Therefore, a total of 60 dynamic simulations were executed, before selecting the most suitable structural solution. HHT-α was chosen as the direct time integration method to carry out the simulations. The behaviour of the different structural alternatives were analysed during 7 seconds. The accelerograms used presented values of ground acceleration at every 0.01 seconds and this time was chosen as the integration time step.

Each simulation demanded a total of 350 MBytes of RAM and generated 126 KBytes of output results. For each structural member, the output les indicated if its stresses and deformations obtained overcame the maximum values according to the construction standard employed, and therefore showing if the member dimension must be changed or not.

4.1. Computational Infrastructure and Execution Results

The Grid deployment that we used for the execution of this structural case study is composed of three clusters of PCs belonging to our research group. The Globus Toolkit 2.4.3 was installed in all

Table 1
Summary of the local Grid deployment and distribution of the simulations in the testbed.

Machine	Computational Nodes	Memory	Avail. Nodes	Simulations
RAMSES	10 Dual Pentium III 866 Mhz	512 MB	10	10
KEFREN	20 Dual Xeon 2.0 Ghz	1 GB	20	20
ODIN	55 Dual Xeon 2.8 Ghz	2 GB	52	30

the machines except Ramses, which runs the LCG-2 middleware and is part of the Biomed Virtual Organisation which will be explained later on.

Table 1 summarises the testbed (i.e., the set of computational resources) employed and the task allocation result. The table indicates, for each machine involved, its principal capabilities, the number of computing nodes that were available just before that scheduling procedure started, together with the number of simulations allocated to it.

The global execution time of the whole structural case study on this Grid was 33 minutes, since the scheduling procedure started until the output data of the last task was retrieved. Later analysis revealed that the executions at Ramses machine lasted 3 times longer than those executed at Odin. This caused the scheduling procedure to wait a few minutes for the executions at Ramses to finish, what could have probably been shortened with a different task allocation policy that had assigned more simulations to the free nodes of Odin cluster. A typical execution model in an engineer studio would involve the sequential simulation of the case study on just one computer. This kind of execution, on one node of Odin cluster, required a total 495 minutes (8.25 hours), what represents a speedup of 15 when executing the 60 structural simulations.

5. Cardiac Case Study

Myocardial ischemia is a condition caused by oxygen deprivation to the heart that can result in an angina. During myocardial ischemia, the extracellular potassium concentration increases in a triphasic pattern, an initial early increase, a constant phase, and a late increasing stage [9]. Other cellular characteristics such as the intracellular concentration of ATP and ADP, and the sodium and the calcium currents that traverse the cell membrane, also vary during this pathology.

In order to study the effects of various degrees of ischemia, a cardiac case study composed of multiple parametric simulations was executed. It analysed the influence, in the electrical propagation, of different degrees of ischemic conditions that take place during the first 10 minutes from the onset of a myocardial ischemia. This case study was originally executed on a restricted Grid deployment, composed of machines from our research group and from another university at Spain, with a naive Globus-based Grid prototype. A fully detailed description of it, which requires the execution of 21 parametric simulations, can be found in a related paper [2]. However, the simulation time has been reduced from 80 ms., in the original case study, to 20 ms, as previous executions revealed that the important information for this particular study appeared within the first 20 ms.

5.1. Computational Infrastructure

The execution of this case study has been carried out in a large deployment. The LCG[2] project aims at the development of the computing infrastructure for the simulation, processing and analysis of the data provided by the Large Hadron Collider (LHC), the most powerful particle accelerator

[2]LHC Computing Grid Project. http://lcg.web.cern.ch

Table 2
Initial state of the resources and distribution of the cardiac simulations in the testbed, for each machine. The number in parentheses indicates the number of nodes involved in the execution.

Machine	Country	Nodes	Memory	Av. Nodes	Simulations
RAMSES	Spain	10 Dual Xeon 866 Mhz	512 MB	8	2 (7 p.), 1 (2 p.)
CNAF-INFN	Italy	8 Dual Xeon 2.4 Ghz	512 MB	7	7 (1 p.)
IN2P3-1	France	112 Dual Xeon 3 Ghz	4 GB	42	2 (1 p.)
IN2P3-2	France	58 Dual Pentium IV 1 Ghz	512 MB	28	8 (1 p.)
BA-INFN	Italy	84 Dual Xeon 2.4 Ghz	881 MB	32	1 (1 p.)

which is being constructed at CERN. The LCG-2 middleware, developed in the mentioned project, was the starting point of EGEE[3], a European Union funded project that aims at developing a service Grid infrastructure available to scientists 24 hours a day.

The LCG-2 middleware is based on the Globus Toolkit 2, providing additional enhancements and functionalities to support the execution of the experiments to be performed in the eld of physics. Within the framework of both the LCG and the EGEE projects, a large testbed composed of machines is available for execution. This testbed currently consists of more than 9500 CPUs, from 112 sites, across 31 countries, with a total storage capacity of 4 Petabytes, representing the largest computational deployments devoted to Grid Computing in science.

Although the LCG project was originally devoted to physics, the EGEE project fosters the migration of biomedical applications to the Grid. In fact, the *Biomed* Virtual Organisation (VO) has been created with those computational resources belonging to project partners that support biomedical research. It currently consists of more than 2500 processors, from 35 sites, across 13 countries.

5.2. Execution Results

Currently, the submission of parallelised applications with the MPI standard is not supported in the EGEE testbed. Therefore, even though GMarte allows the execution of parallel applications on multiprocessor or clusters of PCs, the simulations on machines belonging to the EGEE testbed will always be sequential, that is, with just one processor. However, the Ramses cluster, although belonging to the EGEE testbed, has been specially con gured to support MPI-based executions.

Table 2 summarises the task allocation process. The table shows, for each machine involved in the scheduling procedure, its corresponding country and its features, the number of computing nodes that were available just before the scheduling process started and the number of simulations allocated. The parentheses indicate the number of processors employed in each execution.

When the executions were performed, the resource discovery listed a set of potential remote machines. The resource ltering phase discarded some of them that were unavailable or stated authorisation problems with the user credentials supplied. Also, the machines that did not satisfy the minimum execution requirements were also discarded. This notably reduced the available computing resources. In addition, several hosts were never selected for execution, mainly due to the large data transfer cost estimated (for example, one machine located in Taiwan).

The total execution time of the case study was 102 minutes. Performing a sequential execution of the case study, one simulation after another on a Pentium IV machine, required a total of 1569 minutes. Using a cluster of such PCs, performing executions with 8 processors one after another, lasted

[3]Enabling Grids for E-sciencE. http://eu-egee.org

for a total 221 minutes. This represents a speedup of 15.38 and 7.1 respectively when executing the 21 simulations on the Grid infrastructure. In terms of global execution time, the Grid computing approach enabled to reduce the simulation time of the case study from more than one day to less than two hours. This increases the research productivity as more simulations per day can be achieved.

6. Discussion: From Local to Global Grid

As shown, the solution adopted to accelerate the execution of the case studies have been to perform multiple concurrent simulations on the machines of a Grid infrastructure. Under an ideal situation, we could assume that the Grid would offer enough computing resources for all the executions to proceed simultaneously. Moreover, the underlying middleware would have a negligible cost and data movements among the resources of the Grid would be instantaneous. With these assumptions, the global time of executing a case study would be reduced to the time required by the slowest task. However, as ideal conditions can not be applied to distributed infrastructures such as a Grid, we have to cope with these drawbacks.

When moving from a local Grid to a global Grid, many problems arise. First of all, the lower bandwidth and the increased latency of the communication networks make the data transfer cost a crucial parameter that needs to be considered when selecting a machine for execution. GMarte implements a resource selection policy that considers resource proximity by computing the bandwidth available, for each data transfer, from the local to the remote machine, which is averaged among all the data movemements with this host to provide valuable feedback to the metascheduler.

Second, the resource selection phase, involves a huge amount of communications to gather all the information required from each candidate host in order to select the most appropriate one for the execution of a given task. GMarte implements a *cache* component that stores the static attributes of each machine (i.e. Operating System, CPU architecture, etc.) so that no unnecessary queries and communication are performed. We have also uncoupled the most important stages of remote task execution (resource selection, le stage in, execution and le stage out), allowing multiple threads to independently perform these phases for different tasks. This approach enables to concurrently perform multiple task execution, thus accelerating the global procedure of scheduling.

Finally, a Global Grid subject to different management policies, and an obvious computational diversity, is very prone to distinct kind of failures. In GMarte, failures are managed at different levels. A data transfer failure is retried several times before notifying the error to the upper software layers. Also, failures during the execution of a task are detected by the scheduler. In both cases, the application is re-scheduled for execution in a different machine. A failure in a resource is also managed by re-scheduling the tasks as well as marking the resource as failed. A *resource resurrector* component periodically investigates if the resource is available again for execution.

7. Conclusion

In this paper, we have described the usage of Grid Computing in two different applications, one performing the 3D structural dynamic analysis of buildings and the other simulating the electrical activity in cardiac tissues. A common software layer has been employed, called GMarte, which enables to perform fault-tolerant metascheduling over Globus-based Grid infrastructures.

We have performed the execution of a structural case study on a local Grid and a cardiac case study on part of a large-scale Grid available in the framework of the EGEE project. Substantial reductions in the global execution times of both case studies have been achieved by performing multiple concurrent executions on the distributed infrastructures. Finally, we have also discussed some

important aspects that must be considered when employing a large Grid. As said in the introduction, the Grid is still in the early stages, and much research and developments have to be performed in order to turn the Grid into an easy-to-use technology for its application in different scienti c elds. However, important bene ts can be obtained, as shown, for computationally intensive applications.

Acknowledgements

The authors wish to thank the nancial support received from the Spanish Ministry of Science and Technology to develop the GRID-IT project (TIC2003-01318), the Conselleria de Empresa, Universidad y Ciencia - Generalitat Valenciana for the GRID4BUILD project (GV04B-424), and the Commission of the European Communities, Directorate-General Information Society, Research Infrastructures, for the EGEE project (IST-2003-508833). The authors would also like to thank the Instituto Nazionale di Fisica Nucleare (INFN), the Institute National de Physique Nucleaire et de Physique des Particles (IN2P3) for providing the computional resources, in the framework of the EGEE project, on which some executions were performed.

References

[1] J. M. Alonso, J. M. Ferrero (Jr.), V. Hernández, G. Moltó, M. Monserrat, and J. Saiz. Computer simulation of action potential propagation on cardiac tissues: An efficient and scalable parallel approach. *Parallel Computing: Software Technology, Algorithms, Architectures and Applications, included in series: Advances in Parallel Computing*, 13:339–346, 2004.

[2] J. M. Alonso, J. M. Ferrero (Jr.), V. Hernández, G. Moltó, M. Monserrat, and J. Saiz. Three-dimensional cardiac electrical activity simulation on cluster and grid platforms. In *Proceedings of the Vecpar 2004 International Conference*, pages 485–498, 2004.

[3] J. M. Alonso and V. Hernández. 3D structural dynamic analysis with parallel direct time integration methods. In *Proceedings of the The Tenth International Conference on Civil, Structural and Environmental Engineering Computing*, 2005.

[4] J. M. Alonso, V. Hernández, and G. Moltó. Enabling high level access to grid computing services. *ERCIM News. Special: Grids: The Next Generation*, 59:42–43, 2004.

[5] J.M. Alonso, V. Hernández, and G. Moltó. An object-oriented view of grid computing technologies to abstract remote task execution. In *Proceedings of the Euromicro 2005 International Conference*, pages 235–242, 2005.

[6] V. Berstis. *Fundamentals of Grid Computing*. IBM Redbooks Paper, 2002.

[7] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, 1997.

[8] T.C. Fung. Numerical dissipation in time-step integration algorithms for structural dynamic analysis. *Progress in Structural Engineering and Materials*, 5:167–180, 2003.

[9] K. Sakamoto, J. Yamazaki, and T. Nagao. Diltiazem inhibits the late increase in extracellular potassium by maintaining glycolytic atp synthesis during myocardial ischemia. *Journal of Cardiovascular Pharmacology*, 30(4):424–430, October 1997.

[10] G. von Laszewski, I. Foster, J. Gawor, and P. Lane. A Java commodity grid kit. *Concurrency and Computation-Practice & Experience*, 13(8-9):645–662, July 2001.

[11] S. Wilkinson and D. Thambiratnam. Simplified procedure for seismic analysis of asymmetric buildings. *Computers and Structures*, 79:2833–2845, July 2001.

Parallel Compression of 3D Meshes for Efficient Distributed Visualization

Andrea Clematis[a], Daniele D'Agostino[a], Vittoria Gianuzzi[b]

[a]IMATI-CNR, Genova, {clematis,dago}@ge.imati.cnr.it

[b]DISI, Univeristy of Genova, gianuzzi@disi.unige.it

In a distributed visualization environment transmission time is dominant because of the amount of data to be moved and the limitations of available bandwidth. In this paper we address the problem to speed up the compression operation of large Triangulated Irregular Networks (TIN) using commodity clusters. In our case TINs represent isosurfaces extracted from volumetric data sets. The proposed parallel compression algorithm is based on Edgebreaker [1], one of the most powerful connectivity compression algorithm, and it exploits mesh partitioning produced during the parallel isosurface extraction operation. In this way a high speed-up of the compression module and a considerable improvement of the visualization system are obtained.

1. Introduction

Nowadays large collections of 3D and volumetric data are available, and many people with expertise in different disciplines access these data through the Web. Isosurface extraction [2] is a basic operation that permits to implement many types of queries on volumetric data. The product of the isosurface extraction operation is a Triangulated Irregular Network (TIN) often containing a huge number of triangles, depending on the original data set and on the requested isovalue.

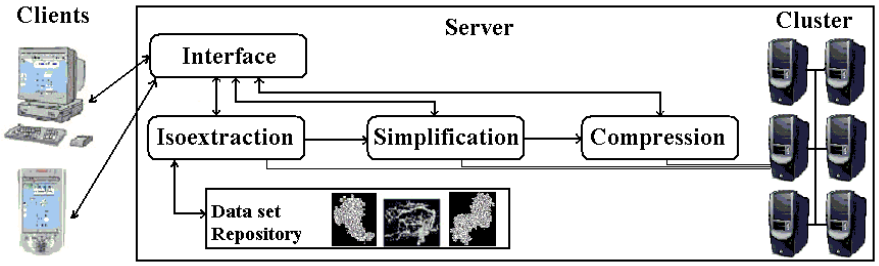


Figure 1. A data processing pipeline for remote visualization

In [3] a typical scenario is presented, where a client accesses a Web server in order to study a 3D data set, for example to visualize protein surfaces in order to define correlation between different macromolecules [4]. The computational pipeline executed on the server consists of an isosurface extraction step [5], that may be followed by a simplification step [6] for progressive visualization, and a compression step to reduce the amount of transferred data as represented in Figure 1.

Reducing the amount of transmitted data is of paramount importance for remote visualization, in particular in a Grid environment [7]. The interconnection network in fact may have variable characteristics, but even using advanced interconnection technologies, the available bandwidth may

represent the bottleneck of a visualization system [8], due to data size. Our computational pipeline doesn't make exception, because extracted isosurfaces can be very large.

The pipeline execution may benefit of parallel computing at different levels, and may have an adaptive behaviour in order to provide the best suited configuration depending on the specific visualization task. In this paper we deal with the parallelization of the compression step using a cluster of COTS PCs. In the pipeline configuration the exploitation of the data partitioning derived from the previous steps makes its execution particularly efficient, because the adopted domain partitioning strategy is finalized to obtain the best speed up for the whole pipeline [9]. However the proposed parallel algorithm can be used in different software configurations provided that it is combined with a suitable TIN partitioning algorithm.

The paper is organized in the following way: in Section 2 different approaches to compress triangular meshes are shortly described, in Section 3 Edgebreaker compression algorithm is introduced, and in Section 4 the parallel compression algorithm and the mesh partitioning strategy are outlined. In Section 5 experimental results are discussed, followed by conclusions and future works in Section 6.

2. Triangular Mesh Compression

A triangular mesh is defined by its geometry and connectivity. The *geometry* corresponds to the vertices, represented by their coordinates (with possibly other information as normal, textures ...), while the *connectivity* is represented by the triangle-vertex incidence relation.

For this reason mesh compression is considered as composed by two distinct operations, *geometry and connectivity encoding*.

Several efficient general purpose techniques were proposed to deal with geometry encoding. They are mostly based on three operations: quantization, prediction, and entropy coding. On the contrary for connectivity encoding the proposed techniques present more or less good results depending on data characteristics.

In [10] the different connectivity compression techniques are classified as *single-rate* and *progressive*. In the first case the mesh is completely encoded before the transmission and decoded after the complete reception. In the second case a coarse mesh is initially received and decoded, then it is progressively refined.

It is possible to further subdivide single-rate algorithms between *lossy* and *lossless*. Lossy algorithms as [11] after the decompression do not recreate exactly the original meshes as lossless ones do. The lossy approach may be useful when the original connectivity does not need to be preserved, so that the triangle mesh may be re-sampled to produce a more regular approximating mesh.

Progressive techniques are useful to provide the client with the possibility to visualize meshes at different levels of details. This can be useful in particular for large meshes, because the visualization of surfaces made by several millions triangles is difficult, if not impossible, even using advanced workstations. In our visualization pipeline we face the problem with the simplification component, that reduces the number of triangles, for example by merging the planar ones. Furthermore, we do not make assumption on the importance of the original connectivity. For these reasons we decided to adopt a single-rate, lossless compression algorithm, and among the interesting proposals belonging to this class as [12, 13], we chose to parallelize "Edgebreaker" [1], one of the most performant single-rate, lossless connectivity compression algorithm.

An in-depth study of this class of algorithms is beyond the goal of this paper, and the interested readers may refer to [14, 10] for a survey and to [15] for a comparison of different algorithms with Edgebreaker.

Edgebreaker is one of the best connectivity compression algorithm, because it allows to compress the connectivity of a manifold triangular mesh homeomorphic to a sphere using 1.84 bit in the worst case for each triangle. This algorithm was afterward improved to threat efficiently meshes with handles [16] and bounding loops [17] as it is the case of most isosurfaces, thus we decided to parallelize this algorithm for our computational pipeline.

Our parallel implementation of Edgebreaker permits to achieve a high speed-up, and at the same time provides a solution to process large meshes by using the aggregate memory of a COTS cluster. In fact, one key problem, common to most of the compression techniques, is that they work in core, then they are not able to process very large meshes. In [18], an out-of-core version of the Touma and Gotsman algorithm [12] is proposed, while at the best of our knowledge there are no previous works about parallel version of TIN compression algorithms.

3. The Edgebreaker Compression Algorithm

The input of the Edgebreaker algorithm is a manifold, orientable meshes that may contain handles and bounding loops.

The algorithm is a finite state machine: given a starting triangle it traverses all the other triangles one at a time along a spiral path. Each visited triangle and its vertices are marked. At the beginning, only the start triangle and its three vertices are marked. The decision of which triangle adjacent to the current one will be visited in the next step is based only on local information and five rules. These rules correspond to label each triangle with one character between ‘C’, ‘L’, ‘E’, ‘R’, ‘S’. The ‘C’ and ‘S’ cases correspond to triangles having neither the left nor the right triangles marked. The difference is that for the ‘C’ case one of the triangle vertices is unvisited. The ‘L’ and ‘R’ labels are used when respectively only the left triangle or the right triangle has already been visited. The ‘E’ case arises when all the neighbour triangles are already marked.

The complete connectivity is encoded considering the label associated to each triangle in the visiting order. The result is a string over this five-symbol alphabet named *clers* string, as exemplified in Figure 2.

Edgebreaker is a pure connectivity compression algorithm. However it is possible to combine it with the compression of the geometry, as described in [16]. The vertex coordinates are encoded in correspondence of ‘C’ triangles using the *parallelogram rule*. The result of the geometry encoding is a sequence of corrector factors plus few coordinates encoded explicitly. This sequence is called *delta* string. We consider as the output of this “complete” version of Edgebreaker both the strings representing the connectivity and the geometry. The size of these strings normally is further reduced applying a general purpose entropy encoding technique.

If a surface is composed by several connected components, that is the case for many isosurfaces, the algorithm processes each component independently, and generate a pair of (clers, delta) strings for each component.

4. The Parallel Compression Algorithm

The presented compression algorithm is a good candidate for parallel processing, in fact most of the operations performed during this process are local. In the case of different connected components we may process them in parallel in a straightforward way, since their processing is independent as noticed above. However, we may incur in load unbalancing due to the different number of triangles of each component. In order to deal with general isosurfaces, that may contain a single large component, and to ensure load balancing, we should find out a suitable mesh partitioning strategy

and verify the correctness and the quality of the algorithm. It is useful to consider initially load balancing, and then discuss correctness and quality of the parallel algorithm.

In order to balance the cost of parallel execution of Edgebreaker across different processors, it is necessary and sufficient to balance the number of triangles contained in the partitions of the mesh distributed among processors. In fact the encoding cost of the clers string is proportional to the number of triangles, while the encoding of the delta string is proportional to the number of vertices, but, following the Euler rule, the number of triangles and the number of vertices are related. To divide a TIN mesh with an approximate equal number of triangles for each partition is in general a costly operation because of the irregular structure of the mesh [19].

In our case, we exploit the information about triangles distribution across the original volumetric data set collected during the first phase of the isosurface extraction process. This information is of paramount importance to obtain a load balanced partitioning of the isosurface, that is exploited during the simplification and compression steps. The adopted partitioning strategy is based on the division of the surface along the z-axis using orthogonal slices. The information collected during the first phase of isoextraction permits to assign a processor the portion of the isosurface included between two slices in such a way that each processor will handle about the same number of triangles, and a minimal set of borders.

Let us consider now the algorithm correctness and the quality of the produced output.

The parallel algorithm is correct if the mesh generated after the decompression is equivalent to the original mesh. Our parallel implementation provides the same result of the sequential algorithm, for each partition of the mesh, provided that a unique bounding box for the quantization of the coordinates of the vertices is used by the different processes, in order to avoid topological errors after the geometry decompression. This requires a unique data exchange among processes before the quantization operation.

The quality of the algorithm is firstly measured by the achieved compression rate. Mesh partition-

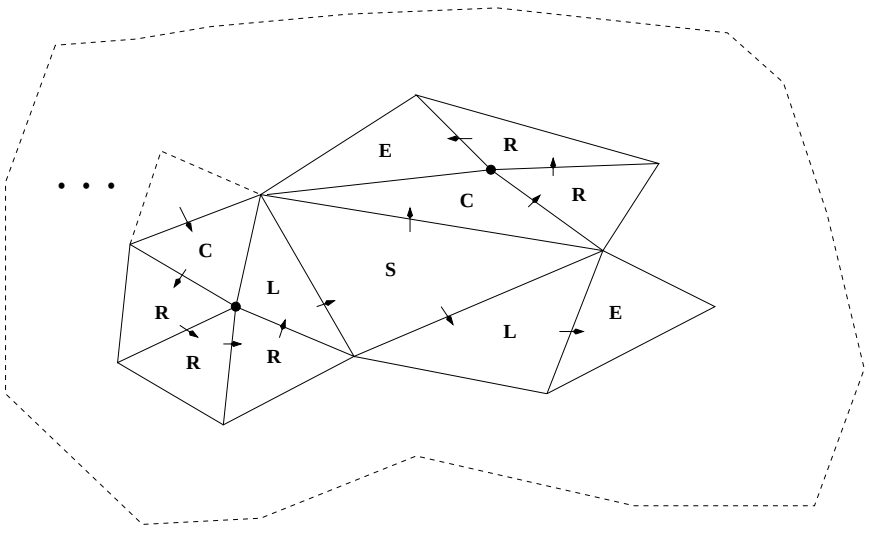


Figure 2. An example of connectivity encoding using Edgebreaker. The dashed lines represent the part of the mesh previously visited and encoded. Not visited vertices are denoted with black circles.

ing introduces new borders that do not exist in the original mesh. The new borders cause an increase in the size of the compressed mesh, mainly because of border vertices replication. Thus the parallel algorithm achieves a worst compression rate. Quantitative evaluation are provided in the next Section. In order to avoid the replication, border vertices could be kept in a shared data structure and the quantization and entropy reduction process, as well as the decompression operation, should be suitably modified. On the base of the analysis of trade-off between the saved space and increment in the computational cost we decided to accept to pay the fee of a reduced compression rate, while keeping the high scalability of the parallel algorithm.

Despite the reduced compression rate attained, the parallel algorithm has many merits:

- The parallel algorithm is highly scalable and an almost linear speed-up is achievable in many cases as shown by experimental results. This point has a large influence on the performance of the whole visualization pipeline.
- The parallel algorithm, using the aggregate memory available on the cluster, increases in a significative way the size of meshes that can be compressed. To this purpose the parallel approach seems to be more competitive with respect to the use of out-of-core algorithms because of the speed-up in the compression process. Moreover the scalability of the parallel algorithm makes it possible to handle meshes of arbitrary size, provided that enough processing units are available.
- As a side effect, the decompression operation can be afforded by almost any client at a reasonable cost, because of the reduced size of each partition.
- Despite its simple parallel implementation, the compression module is of paramount importance for the efficient and effective use of the visualization pipeline.

These merits are supported by the experimental results provided in the next Section.

5. Experimental Results

Sequential computing times have been collected using a Linux PC equipped with a 2.66 GHz Pentium processor, 512 MB of Ram and two EIDE disks interfaced in RAID 0. Parallel computing times have been collected using a cluster of 16 PCs with the previous characteristics, interconnected through an Ethernet - Gigabit switch and having input data sets stored in a PVFS 1 parallel filesystem. The sequential and parallel algorithms are based on the code freely available in [20] and [16], both are implemented in C and the MPICH library is used for parallelization.

Table 1
This table summarizes I/O data volumes of the isosurfaces extracted for different data sets and isovalues and the size of data compressed with the sequential implementation of Edgebreaker

Data set - isovalue	Input size	Triangles	Output Size	Edgebreaker
Bonsai - 2	16 MB	3,896,986	67 MB	5.7 MB
Frog - 75	30 MB	2,655,552	45 MB	3.7 MB
Xmastree - 180	499,5 MB	4,514,539	78 MB	5.7 MB
Xmastree - 52	499,5 MB	21,719,037	374 MB	31 MB
VisFemale - 1000	867 MB	69,600,216	1.2 GB	N.A.

In Table 1 the characteristics of meshes used in the tests are presented. These isosurfaces are extracted from Computed Tomography scans of a bonsai, a frog, a Christmas tree, and a female cadaver. The Christmas Tree data set was generated from a real world Christmas Tree by the Department of Radiology, University of Vienna and the Institute of Computer Graphics and Algorithms, Vienna University of Technology. The female cadaver is an anatomical data set developed under a contract from the NLM by the Departments of Cellular and Structural Biology, and Radiology, University of Colorado School of Medicine.

We can see that with Edgebreaker the size of compressed data is about 12% of the original size using a quantization on 16 bits. The compressed data size for “VisFemale - 1000” is not available because the mesh is too large for a sequential implementation.

Table 2
This table summarizes the times in seconds of the sequential and parallel version of Edgebreaker algorithm for the isosurfaces described in Table 1. In brackets the speed up values.

Data set - isovalue	Sequential	4 procs	8 procs	16 procs
Bonsai - 2	10.28	2.85 (3.6)	1.42 (7.6)	0.66 (15.5)
Frog - 75	6.81	1.84 (3.7)	0.87 (7.8)	0.52 (13.1)
Xmastree - 180	10.73	2.92 (3.7)	1.43 (7.5)	0.68 (15.8)
Xmastree - 52	796.30	22.99 (-)	6.94 (-)	3.42 (-)
VisFemale - 1000	N.A.	N.A.	N.A.	19.21 (N.A.)

In Table 2 execution times in seconds for the sequential and the parallel compression algorithm together with speedup values (in brackets) are presented. We used 4, 8 and 16 processors to execute the parallel algorithm.

The dataset “VisFemale - 1000” is manageable only using 16 processes, with an execution time of about 19 seconds. With dataset “Xmastree - 52” we got using 16 processors a speed up of about 232. In fact to treat this mesh, the sequential algorithm has to use the virtual memory of the system, with a considerable performance degradation due to frequent page faults. Thus we do not consider the obtained super linear speed-up for this and similar cases. However these results emphasize the importance of parallel compression.

As said before, to evaluate the effectiveness of our parallelization strategy it is necessary to take into account both the speed up achieved and the quality of the results, represented by the size of the entropy encoding of the “delta” and the “clers” strings.

Table 3
This table compares the size of the compresses data resulting from the sequential algorithm, the parallel algorithm executed using 16 processes, and GNU Gzip

Data set - isovalue	Sequential EB	Parallel EB	Gzip
Bonsai - 2	5.7 MB	6.3 MB	23 MB
Frog - 75	3.7 MB	4 MB	16 MB
Xmas tree - 180	5.7 MB	6.1 MB	28 MB
Xmas tree - 52	31 MB	32 MB	134 MB
VisFemale - 1000	N.A.	112 MB	443 MB

In Table 3 the incidence of the duplicated boundary vertices is presented in the worst case, when each isosurface is partitioned among 16 processors. As we can see, the data size is increased at most of 10% . In practice, the degradation due to vertex duplication has not an heavy impact from the result size point of view, while the speed up achievable is nearly linear.

In the same table we present a comparison between the size of the data compressed using Edgebreaker and a general purpose technique, GNU Gzip, to demonstrate how a specific compression technique is more effective.

To better appreciate the effectiveness of the use of the parallel compression we consider the total time needed to transmit isosurface from the server to a client in three different cases:

1. the transmission is done without compression;
2. sequential compression is adopted, and decompression is executed on the client side;
3. parallel compression is adopted, and decompression followed by a mesh sewing is executed by the client.

The client is a Linux PC with the same characteristics of the the node of the cluster, while the transmission is on a geographic network connecting our institute in Genoa with the Institute for Biotecnologies of CNR in Milan, and the network bandwidth is of about 143KB/sec.

The measured total time for the Xmas tree - 52 is 44min:37sec. without compression, 17min:36sec. with sequential compression and 4min:31sec with parallel compression.

6. Conclusion and Future Works

The main contribution of this paper is a data parallel compression algorithm for TINs based on the Edgebreaker connectivity compression algorithm.

Despite the simple approach, our algorithm presents two advantages. The first is the high speed up obtainable, because each process encodes its data independently. The second is the possibility to efficiently compress meshes of arbitrary size using a cluster of PCs with a sufficient amount of aggregate memory. The decompression remains a sequential operation, but is able to manage the produced results, because they are provided in a partitioned form.

A drawback is represented by the degradation of the quality of the results that, for a compression algorithm, is represented by the size of the compressed data. This increment is of about 10% so, considering also the achieved near linear speed up, we can conclude that the proposed algorithm represents a good compromise between the speed up achievable and the quality of the results.

We plan to develop an ad-hoc mesh partitioning algorithm and to improve the algorithm in order to reduce the impact of duplicated vertices.

7. Acknowledgments

This work has been supported by MIUR programme L.449/97-00 High Performance Distributed Computing Platform, and by FIRB strategic project on Enabling Technologies for Information Society, Grid.it.

References

[1] J. Rossignac: Edgebreaker: Connectivity compression for triangle meshes. IEEE Transactions on Visualization and Computer Graphics, Vol. 5, No. 1, pp. 47-61. 1999.

[2] W.E. Lorensen, H.E. Cline: Marching Cubes: a high resolution 3D surface reconstruction algorithm. Computer Graphics, Vol. 21, No. 4, pp. 163-169. 1987.

[3] A. Clematis, D. D'Agostino, W. De Marco and V. Gianuzzi: A Web-Based Isosurface Extraction System for Heterogeneous Clients. Proceedings of the 29th Euromicro Conference, IEEE Computer Society Press, pp. 148-156. 2003.

[4] I. Merelli, Luciano Milanesi, Daniele D'Agostino, Andrea Clematis, Marco Vanneschi, Marco Danelutto: Using Parallel Isosurface Extraction in Superficial Molecular Modeling. Proceedings of the 1st International Conference on Distributed Frameworks for Multimedia Applications (DFMA 2005), IEEE Computer Society, pp. 288-294. 2005.

[5] A. Clematis, D. D'Agostino, V. Gianuzzi: An Online Parallel Algorithm for Remote Visualization of Isosurfaces. Proceedings of the 10th EURO PVM MPI Conference, LNCS no. 2840, pp. 160-169. 2003.

[6] A. Clematis, D. D'Agostino, V. Gianuzzi, M. Mancini: Parallel Decimation of 3D Meshes for Efficent Web based Isosurface Extraction. Proceedings of the International Conference of Parallel Computation (PARCO 2003), Advances in Parallel Computing No. 13, pp. 159-166. 2004.

[7] P. Heinzlreiter, D. Kranzlmüller: Visualization Services on the Grid: The Grid Visualization Kernel. Parallel Processing Letters, Vol. 13, No. 2, pp. 135-148. 2003.

[8] S. Fisher: Berkeley Lab Proves 10-Gigabit Ethernet Data Transfer is a Reality. Supercomputing Online. 3 Jul. 2003.

[9] A. Clematis, D. D'Agostino, V. Gianuzzi: Load Balancing and Computing Strategies in Pipeline Optimization for Parallel Visualization of 3D Irregular Meshes. Accepted to the 12th EURO PVM MPI Conference. Sorrento (Naples), ITALY, September 18/21, 2005

[10] P. Alliez and C. Gotsman: Recent Advances in Compression of 3D Meshes. Proceedings of the Symposium on Multiresolution in Geometric Modeling, 2003.

[11] M. Attene, B. Falcidieno, M. Spagnuolo and J. Rossignac: SwingWrapper: Retiling Triangle Meshes for Better Compression. ACM Transactions in Graphics, Vo. 22, No. 4, pp. 982-996, 2003.

[12] C. Touma and C. Gotsman. Triangle Mesh Compression. Graphics Interface 98 Conference Proceedings, pp. 26-34, 1998.

[13] S. Gumhold and W. Straßer. Real time compression of triangle mesh connectivity. Proceedings of SIGGRAPH '98, pp. 133-140, 1998.

[14] C. Gotsman, S. Gumhold, and L. Kobbelt: Simplification and Compression of 3D Meshes. Tutorials on Multiresolution in Geometric Modelling, A. Iske, E. Quak, and M.S. Floater (eds.), Springer-Verlag, pp. 319-361, 2002.

[15] J. Rossignac: 3D Mesh Compression Chapter in The visualization handbook, C. D. Hansen and C. R. Johnson (eds.), Academic Press, 2004.

[16] A. Szymczak, D. King, J. Rossignac: An Edgebreaker-based efficient compression scheme for regular meshes. Computational Geometry, Vol. 20, No. 1-2, pp. 53-68. 2001.
The code is available at http://www.cc.gatech.edu/fac/Andrzej.Szymczak/eb.html

[17] T. Lewiner, H. Lopes, J. Rossignac, A. Wilson-Vieira: Efficient Edgebreaker for surfaces of arbitrary topology. Proceedings of 17th Brazilian Symposium on Computer Graphics and Image Processing, (SIBGRAPI'04), pp. 218-225. 2004.

[18] M. Isenburg, S. Gumhold: Out-of-core compression for gigantic polygon meshes ACM Transactions on Graphics, vol 22, no. 3, pp. 935-942, 2003.

[19] C. Walshaw, M. Cross, and M.G. Everett, Parallel Dynamic Graph Partitioning for Adaptive Unstructured Meshes Journal of Parallel and Distributed Computing, 47, pp. 102-108, 1997.

[20] Edgebreaker source code: http://www.gvu.gatech.edu/ jarek/edgebreaker/eb/

A parallel implementation of a 3d reconstruction algorithm for real-time vision.

J. Falcou[a],J. Sérot[a], T. Chateau[a], F. Jurie[b]

[a]LASMEA, UMR6602 UBP/CNRS, Campus des Cézeaux, 63177 Aubière, France.

[b]GRAVIR, INRIA/CNRS, 55 avenue de l'Europe, Montbonnot 38330, FRANCE

1. Introduction

Artificial vision requires of large amount of computing power, especially when operating on the fly on digital video streams. For these applications, real-time processing is needed to allow the system to interact with its environment, like in robotic applications or man/machine interfaces. Two braod classes of solutions have been used to solve the problem of balancing application needs and the constraints of the real-time processing: degrading algorithms or using dedicated hardware architecture like FPGA or GPU. These strategies were effective because of the specific properties of the images and the structure of the associated algorithms. However, the constant and fast progression of general purpose computers performance makes these specific solutions less and less interesting. Development time and cost now plead in favor of architectures based on standard components. During the last ten years, the use of these solutions increased with the generalization of *clusters* made up of off-the-shelf personal computers. But this type of solution has been rarely used in the context of complex vision applications operating on the fly. This paper evaluates this opportunity by proposing a cluster architecture dedicated to real-time vision applications. We describe the hardware architecture of such a solution – by justifying the technological choices carried out on the application requirements and the current state of the art – then the associated software architecture. The validity of the approach is shown with the description and performance evaluation of a real-time 3D reconstruction application.

2. Architecture Design

2.1. Hardware architecture

The hardware architecture of the cluster is shown on fig.1. The current configuration includes 14 compute nodes. Each node is a dual-processor G5[1] running at 2 GHz and with 1Gb of memory. Nodes are interconnected with Gigabit Ethernet and provided with digital video streams, coming from a pair of digital cameras, by a *Firewire IEEE1394a* bus [2]. This approach allows simultaneous and synchronized broadcasting of input images to all nodes, thus removing the classical bottleneck which occurs when images are acquired on a dedicated node and then explicitly broadcasted to all other nodes. The Ethernet network bandwidth is then fully available for application-level message-passing.

[1]Apple XServe Cluster Node

[2]In isochronous mode, at 400Mb/s

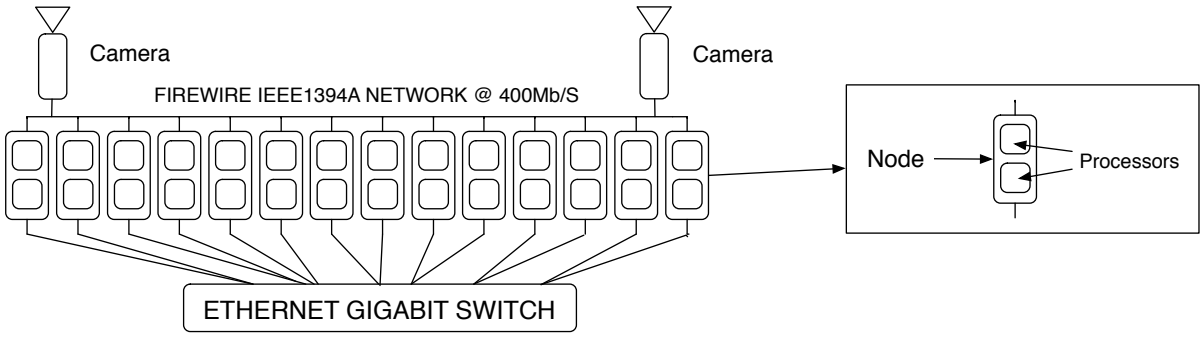


Figure 1. Cluster architecture

The following motivated the choice of the G5 processor :

- **The presence of the Altivec extension [7].** This extension can provide speedups in the range of 2-10 for a large class of low to mid-level image processing operations [5]. Without it, the size of the cluster needed for obtaining the required speed-ups (up to 50) would have been prohibitive (in terms of power consumption and cost).
- **The native support of Gigabit Ethernet network**. Gigabit Ethernet offers one of the best cost/performance ratio. More expensive solutions like MyriNet or InfiniBand have been considered, but their high cost will have reduced the actual number of node available for the cluster.

The suggested architecture is sufficiently open to enable it to benefit directly from technological progress, while supporting mainstream programming models and tools.

2.2. Software architecture

The software architecture is a layered one :

1. At the lowest level are the ***Firewire* drivers** handling the transfer of images from the cameras to the processor memory (using the DMA). For the programmer, the calls to these drivers are encapsulated within a dedicated library (CFOX[6]) which provides in particular functions for configuring cameras and obtaining a pointer to the most recently acquired frame in memory. *CFOX* also provides a easy way to synchronize video streams (sec. 3.3).
2. The middle layer deals with **parallelism issues**. We use a hybrid three-level parallel programming model, involving a fine-grain SIMD-type parallelism within each processor (Altivec), a coarse grain shared-memory multi-processing model between the two processors of a node and a coarse grain message passing based multi-processing between two processors of distinct nodes. The first level can be exploited using the Altivec native C API [7] or, more easily, using EVE [5], a high-level vectorization library specifically tuned for this extension. The second level is exploited by describing the process running on each node as two concurrent threads using the PTHREAD library. At the third level, the application is decomposed into a set of processes running in SPMD mode and communicating using the MPI library. The use of

PTHREAD to explicitly schedule the execution of threads on the two processors of each node is due to the fact that the lowest level of the software architecture doesn't map correctly onto the dual processor architecture. Each node is viewed as a single *Firewire* node by the driver layer and as a two-processor node by the parallel layer. This duality leads to resources sharing conflicts that can't be handled correctly by MPI. We need to keep a one-for-one mapping between the *Firewire* level and the MPI level and use PTHREAD for this purpose.

3. The upper layer acts as a **high-level programming framework for implementing applications**. This framework, implemented in C++, provides ready-to-use abstractions (skeletons, in the sense of [8]) for parallel decomposition, built-in functions for video i/o and mechanisms allowing run-time modification and parameterization of application-level software modules. Developing specific applications boils down to write independent algorithmic modules and to design an application workflow using those modules. One can develop various workflows using simple text description and dynamically choose which one is used by the application.

Here is a example of how we can write applications using this framework.

```
struct Data : public MPIData<Data>
{
  Frame input;
  Frame output;
};

struct Work : public Task<Work>
{
  bool operator()( Data& d )
  {
    d.output = where(d.input > 127, 255, 0);
    return true;
  }
};

int main(int argc, const char** argv )
{
  Data    d;
  Camera  camera( 30, res640x480, 8 );
  Cluster cluster(argc,argv);

  task_list(RowSplit<Frame>,Work,RowMerge<Frame>) act;
  cluster.task() = (SCM<act>(cluster.root(),cluster.world()));

  camera >> d.input;
  cluster.run(d);

  return 0;
}
```

Figure 2. A simple binary thresholding application using our software architecture

The code shown in figure 2 is a simple example that retrieves video frames from a single camera and relies on a classical scatter-compute-merge skeleton to perform a binary thresholding on them [3]. The first part of this code is the declaration of a class that defines the data we will work on and the operations that will be applied to this data. The `Work` class implements a `()` operator that performs the actual thresholding.This is done by using the EVE function `where` that implements a vectorized `if/then/else` construction. The application itself starts by building a `Data` object, the `Camera` object and the main `Cluster` object. Then tasks that will be run onto this `Cluster` are explicitly scheduled by using one of the built-in parallel skeleton class provided by the application framework. `SCM` is a simple scatter-compute-merge skeleton that splits the input image into slices, dispatches them onto a static number of workers, applies a given operation and merges the result. Here the built-in split and merge classes (`RowSplit` and `RowMerge`) and the `Work` class defined earlier are used. We also explicitely choose the root process of the `SCM` skeleton and the subgroup of nodes that will run the atual tasks. Once the activity is scheduled, video frames are grabbed and dispatched to the `Cluster` object.

3. Realistic application

The proposed platform has been assessed in the context of real-time vision applications in order to show that its dual bus architecture (Ethernet for communication and *Firewire* for video broadcasting) and its three-level parallel programming model (SIMD,SMP,MP) suit the needs of this class of applications in terms of performance and programmability.

3.1. Algorithm description

We implemented the first stages of a 3D reconstruction and tracking chain using stereoscopic vision. The application described here generates set of 3D points; it will obviously be followed by an interpretation stage that we don't cover in this paper. Basically, the algorithm consists in four different steps:

- **Image rectification (RECTIF)** : This stage warps camera frames by matching epipolar lines onto image pixels rows[9]. This simplifies further point matching by limiting search to a single pixel lines.
- **Key-point detection (DETECT)** : point of interest to be used as seed for matching stage are extracted from the rectified stereo pairs by a Harris and Stephen corner detector [4].
- **Key-point matching (MATCH)** : Each 2D key-point from the left image (resp. right image) is matched with a 2D key-point from the right image (resp. left image) by using a maximum likelihood search using a zeros normalized cross correlation (ZNCC) measure.
- **3D reconstruction (BUILD)** : Correctly matched key-point pairs are then triangulated [10] to outputs the 3D coordinates of the corresponding point.

This algorithm was chosen for three reasons: the huge number of operations per frame, the diversity of these operations and the fact it is at the root of numerous applications such as mobile robot navigation, motion capture or human-computer interface. A sample of the results obtained by this algorithm is shown on figure 3.

[3]For clarity purpose, we left out include directives and namespace declaration.

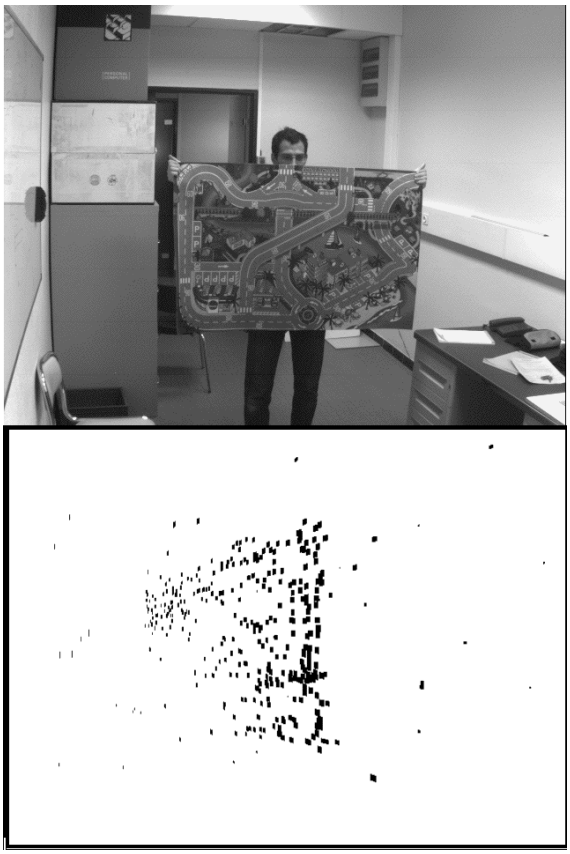


Figure 3. Visual output of the 3D reconstruction algorithm

3.2. Parallelization Strategy

The detection and matching steps both involve regular, iconic processing and have been vectorized using the EVE library [5] in order to take advantage of the SIMD level parallelism offered by the G5 Altivec extension. Parallelization of the rectification, matching and reconstruction steps is done by partitioning the source image into a fixed number of slices and having each processor operating within a slice[4]. A first approach was to use a `Farm` skeleton providing simple workload balancing. In this skeleton, a master node dynamically dispatches images slices to workers that apply each step of the reconstruction algorithm to a slice and outputs a set of 3D points. Finally, sets computed on each workers are merged and displayed. It turned out that the optimal slices size was one slice per processor. We therefore decided to use the `SCM` skeleton shown earlier. Globally, this parallelization scheme only needs two synchronization step : one before grabbing the frame from the camera and one at the end of the merging process.

3.3. Camera Synchronization

To be as accurate as possible, the algorithm must be run onto a stereo pair picturing the same space-time event. If left and right image of the pair are incoherent, the matching and reconstruction will lead to badly reconstructed scene. In practice, this means that acquisition of left and right frames on a single node must be synchronized as well as the broadcasting of images pairs to all node of the

[4]This is possible because the rectification step ensures that matching points will be on the same horizontal line of the rectified image.

cluster. This is done by a two-step synchronization mechanism :

- **A node synchronization step** : The CFOX driver implements a time-stamp verification and is able to provide synchronized frames from any number of cameras. We ensure that when the video frames are acquired, they picture the same space-time event. This synchronization step being a feature of the CFOX driver, users don't have to care about explicitly synchronizing frame acquisition on a single node.
- **A cluster synchronization step** : By using a message passing based synchronization (using `MPI_Barrier`) to force all nodes to wait each other before acquiring the next frame, we force the synchronization of frame acquisition on all nodes. This synchronization has to be explicitely triggered by the user.

4. Results

Preliminary results, obtained on 640×480×8bits video streams, appear in table 1. Computation times for each step of the algorithm and total execution times are given for several values of the processor number[5]. The first column gives the corresponding numbers for a sequential reference implementation measured on a single G5 processor at 2GHz.

Step	Seq	np=2	np=4	np=8	np=16	np=24	np = 28
RECTIF	$246ms$	$139.1ms$	$70.5ms$	$36.1ms$	$19.5ms$	$13.1ms$	$12.6ms$
DETECT	$262ms$	$80.1ms$	$40.5ms$	$20.6ms$	$11.2ms$	$7.1ms$	$6.4ms$
MATCH	$304.2ms$	$180ms$	$91.5ms$	$47.4ms$	$22.4ms$	$13.8ms$	$9.7ms$
BUILD	$180ms$	$100ms$	$53ms$	$27.5ms$	$18.2ms$	$12.0ms$	$9.5ms$
TOTAL	$992.2ms$	$479.2ms$	$244.6ms$	$122.6ms$	$68.2ms$	$42.9ms$	$38.2ms$
FPS	1.02	2.08	4.08	8.15	14.66	23.31	26.17

Figure 4. Preliminary results for the application

Minimum latency is 38.2 ms, obtained with 28 processors. This leads to a maximum throughput of 25.97 images processed per second and perfectly matches real-time vision constraints (cameras are programmed to deliver 30 FPS). The corresponding relative speedup (compared to the sequential reference implementation) is 25.97 (96% efficiency[6]). Two factors can explain these very encouraging results . First, thanks to the automatic broadcasting of images provided by the *Firewire* bus, the application exhibits a high compute *vs* communication ratio (between 0.90 and 0.92 typically). Second, the SIMD parallelization level offered by the Altivec can be efficiently exploited in at least two steps of the application : DETECT and MATCH. Without Altivec, the maximum throughput is only 17 im/s, showing that the presence of the SIMD level provides a noticeable speed-up increase. These preliminary[7] results were obtained with a rather naive implementation. They could be further improved by reformulating certain parts of the involved algorithms to make them more amenable to vectorization / parallelization and by fine tuning several algorithmic and parallelization parameters such has the dimension of the filters used in the MATCH step). For the DETECT and MATCH steps,

[5]The processor handling the display of results is not counted here.
[6]92% if we take into account the processor acting as display.
[7]Obtained with the first full working version of the platform in December 2004.

the speed-up due to AltiVec only is respectively 2,82 and 2,29. This acceleration could be easily increased by changing the data types used for computations, minimizing memory copy and swap or by rewriting some algorithm with a more SIMD-oriented approach than now.

5. Related Work

Running real-time 3D reconstruction applications on clusters has been applied to some variation of our original problem. A proposed implementation of a real-time 3D shape reconstruction algorithm uses N camera connected to N nodes in order to build a visual hull of the scene by a silhouette volume intersection method. Camera synchronization is needed to ensure that each PC/Camera pair is acquiring images at same time and solutions to this synchronization problem are described. The parallelization strategy is based on a simple pipeline in which each algorithm step is allocated to a computer node. Real-time performances are achieved with a small number of nodes and camera. Wu and Matsuyama[1] or Franco and al.[2] expose such results. Those approaches differ from the one presented in this paper in term of parallelization scheme and camera usage. Yoshimoto and Arita[3] describe a Firewire-based PC cluster dedicated to real-time image processing. It consists of a cluster of PCs interconnected by a IEEE-1394a Firewire bus. This bus is used both for transmitting video data from the camera to the processors and for communicating between processors. Although the possibility to obtain real-time performances is demonstrated (on a stereo-based 3D image restoration), sharing the Firewire bus for both video broadcasting and inter-process communication may lead to performance loss or scalability issues.

6. Conclusion

This paper shows that a cluster made of off-the-shelf computers with an efficient video streaming capability and with several parallelism levels meets the constraints of complex real-time vision applications. This was demonstrated by developing a 3D reconstruction application from a stereo video stream. As far as we know, this is the first use of such an architecture to solve real time vision application with this level of complexity. Those results are however preliminary as the cluster has been operational for a few months. Shortly, we plan to finalize the reconstruction/tracking chain to provide a fully functional real time 3D tracking application.

References

[1] Wu X., Matsuyama T. Real-time active 3d shape reconstruction for 3d video. In Proceedings of 3rd International Symposium on Image and Signal Processing and Analysis, pages. 186-191, 2003.

[2] Franco J.-S., Menier C. and Boyer E. A Distributed Approach for Real-Time 3D Modeling.CVPR Workshop on Real-Time 3D Sensors and their Applications, 2004

[3] Yoshimoto H. and al. Real-Time Image Processing on IEEE1394-based PC Cluster. In 15th International Parallel and Distributed Processing Symposium, 2001

[4] Harris C. and Stephens M. A combined corner and edge detector. In 4th Alvey Vision Conference, 1988, pp. 189-192.

[5] Falcou, J., Serot, J - E.V.E., An Object Oriented SIMD Library. In Practical Aspects of High-level Parallel Programming,ICCS 2004(3), pp. 323-330

[6] Falcou, J. - CFOX - A Firewire Camera Driver for OS X `http://wwwlasmea.univ-bpclermont.fr/Personel/Joel.Falcou/software/cfox`

[7] Ollman I. AltiVec Velocity Engine Tutorial. `http://www.simdtech.org/altivec`. March 2001.

[8] Cole M. - Algorithmic Skeletons. In Research Directions in Parallel Functional Programming, Springer-Verlag, 1999.
[9] Fusiello A.,Trucco E., Verri A. - A compact algorithm for rectification of stereo pairs. In Machine Vision and Application, vol. 12, no. 1, pp. 16–22, 2000
[10] Hartley R. I. and Zisserman A. - Multiple View Geometry in Computer Vision. Cambridge University Press - ISBN 0521623049, 2000.

Languages

Using eSkel to implement the multiple baseline stereo application

Anne Benoit[a], Murray Cole[a], Stephen Gilmore[a], Jane Hillston[a]

[a]School of Informatics, The University of Edinburgh, James Clerk Maxwell Building, The King's Buildings, Mayfield Road, Edinburgh EH9 3JZ, UK

We present an application of the *eSkel* skeletal programming library to the multiple baseline stereo problem. We compare its performance to that of a direct MPI implementation of the same algorithm.

1. Introduction

The skeletal approach to parallel programming is well documented in the research literature (see [4,5,7,8] for surveys). It observes that many parallel algorithms can be characterised and classified by their adherence to one or more of a number of generic patterns of computation and interaction. Skeletal programming proposes that such patterns be abstracted and provided as a programmer's toolkit, with specifications which transcend architectural variations but implementations which recognise these to enhance performance. This level of abstraction makes it easier for the programmer to experiment with a variety of parallel structurings for a given application, by enabling a clean separation between structural aspects and application specific details. In the *eSkel* (Edinburgh Skeleton Library) project, motivated by our observations [5] on previous attempts to implement these ideas, we have begun to define a generic set of skeletons as a library of C functions on top of MPI.

This paper describes the first use of *eSkel* on a significant application, the multiple baseline stereo vision problem [6,10]. We begin by providing an overview of *eSkel* and its conceptual basis, before proceeding to a description of the standard multiple baseline stereo algorithm. We describe the facility with which the algorithm can be expressed in *eSkel*, and examine the performance of the resulting programs on a Beowulf cluster and on an SMP, focusing particularly on the overhead incurred by *eSkel* when compared with an explicit MPI version of the same algorithm.

2. Structured parallel programming with *eSkel*

The *eSkel* project [1–3] is an ongoing attempt to investigate the practicality and applicability of skeletal programming systems. Building on previous experiences, its aims (to which we will return in our experimental evaluation) were stated [5] as being to develop a skeletal system which

1. Promotes skeletal programming with minimal **disruption** to the "conventional" parallel programmer's conceptual model.

2. Allows the integration of **ad-hoc** (unstructured) code within an otherwise skeletal program.

3. Accommodates a **flexible** collection of variations on familiar parallel programming idioms.

4. Provides empirical evidence that skeletal programming need not incur significant **performance** penalties (and indeed might even provide performance improvements) judged against equivalent ad-hoc parallel programs.

eSkel's conceptual model and API are based around MPI. This basis, and the fact that the implementation is also built on top of MPI, ensure widespread portability.

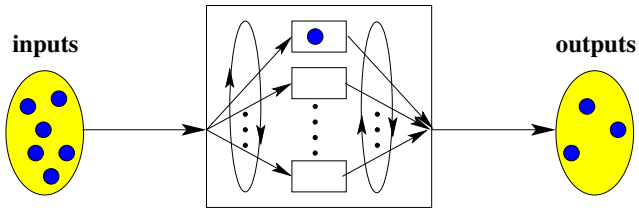


Figure 1. Conceptual structure of the *Deal* skeleton.

In essence, an *eSkel* skeleton is a collective operation, called by all processes within a given MPI communicator group. The call causes the processes to take on the roles of the various components of the chosen skeleton (for example, the stages in a pipeline). It abstracts the implied communications, either completely, leaving the programmer to specify only the activities within each component, or partially, allowing the programmer to control the timing of communications within the spatial constraints imposed by the skeleton (for example, allowing a pipeline stage to consume an input without producing a corresponding output).

Skeleton calls may be nested, either *transiently*, meaning that the inner instantiation is created and exists only between interactions of the outer call, or *persistently*, meaning that the inner call lasts for the duration of the entire outer call (in effect giving a flat skeleton structure combining the two skeletons).

In the application described here, we use the only two of *eSkel*'s collection of skeletons which have so far been fully implemented, the *Pipeline* and the *Deal*. Our pipeline is a straightforward abstraction of the familiar paradigm: a sequence of *stages*, any of which may be internally parallel, processes a sequence of input items to produce a sequence of output items. *Deal*, while less familiar by name, captures another familiar technique, typically applied within bottleneck pipeline stages: the stage computation itself is replicated, with successive inputs dispatched cyclically to the replicas, outputs being merged in the original order into the overall stream passed to the subsequent stage. The technique is only applicable for stages in which no internal state is maintained from one input to the next. Figure 1 sketches the structure of a single *Deal*. More typically, either or both of the input and output streams will be tied to other pipeline stages. Thus, the *Deal* skeleton behaves like a *Farm* with a cyclic allocation of the work to the farmers.

A full discussion of the *eSkel* API is well beyond the space constraints of this paper.

3. The multiple baseline stereo application

The multiple baseline stereo problem [6,9,10] involves measuring depth in a scene with the help of several cameras. The cameras have different baselines, enabling precise distance estimates to be obtained with a stereo matching method. Paraphrasing [9],

> "The input consists of three images, acquired from three horizontally aligned, equally spaced cameras. One image is the *reference image*, the other two are *match images*. For each of 16 disparities $d = 0..15$, the first match image is shifted by d pixels, and the second image is shifted by $2d$ pixels. A *difference image* is formed by computing the sum of squared differences between the corresponding pixels of the reference image and the shifted match images. Next, an *error image* is obtained by replacing each pixel in the difference image with the sum of the pixels in a surrounding 13x13 window. Finally,

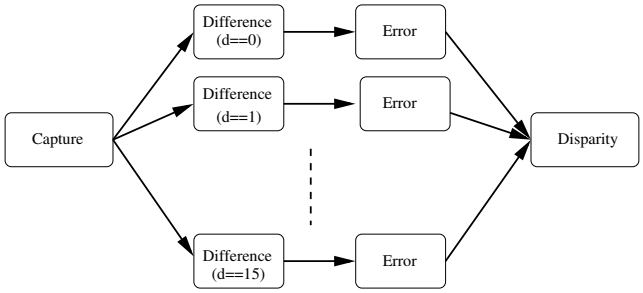


Figure 2. Pipelined structure of the original algorithm.

> the *disparity image* is formed by finding, for each pixel, the disparity that minimises error. We then know the depth of each pixel, which depends on this disparity."

The algorithm naturally lends itself to pipelined and replicated parallelism, with stages for data gathering (or in our case, in the absence of cameras, artificial generation), difference image calculation, error image calculation and disparity image calculation. Each input to the pipeline is a set of three images. For a given triple of images, the difference and error calculations for each of the 16 disparities are independent and so there is scope for concurrent execution, as suggested by figure 2. Finally, we note that the algorithm could apply internal data parallelism to the difference and error computations.

The natural context of the application is in real-time processing of a stream of images. This means that the most appropriate measure of performance is the *throughput* with which such a sequence is processed. We adopt this metric in our performance graphs.

4. Parallel implementation with *eSkel*

For the purposes of our experiments we have chosen to implement a variant of the algorithm described above. As sketched in figure 3, we have built a three stage pipeline, in which the first stage generates images, the second implements all (16 per image set) of the difference, error and disparity computations, and the final stage (which might be a display operation in a real application), simply performs some checking of results. Our artificial inputs are constructed with easy checking of outputs in mind. As noted above, the run-time of the calculations, which is our main concern, depends only upon the size (rather than the content) of the images. Not surprisingly, this structure results in a substantial bottleneck in the second stage. We resolve this with a *Deal* - successive image sets are distributed to successive workers. We ran tests with between one and six replicas.

Stripped of application specific code, the program itself is straightforward. Figure 4 presents the main program (with only simple C declarations omitted to save space). Since images are generated within stage 0, and outputs are not stored, most of the data parameters are redundant (hence all the `NULL`s and `0`s). The `imodes`, `spreads` and `types` parameters to the skeleton call describe the structure of the interfaces between stages.

In this version of the program, we have chosen to use *explicit* communication for data leaving stage 0 and entering stage 2 (line 3). By way of contrast, the workers in the *Deal* will experience these communications in implicit mode. This is specified in two steps. For the stage itself, the interaction mode is given as "devolved", indicating the presence of a persistently nested skeleton.

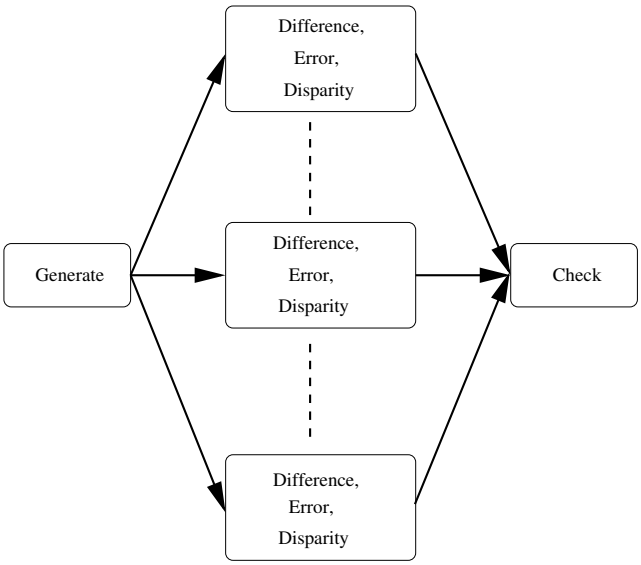


Figure 3. Structure of the algorithm as implemented.

An extract of the code for stage 2 (figure 5) completes the process, by specifying (line 4) *implicit* interaction mode.

Lines 17-19 of figure 5 are the interface between *eSkel*'s data model and the standard C of the computational fragments. Following the MPI approach, the three data arrays are received packed into a single array, accessible through the `data` field of the `eSkel_molecule_t` received by the stage. Rather than wasting time explicitly unpacking the structure, we simply create convenient pointers to the sections corresponding to each array. Lines 32-33 similarly prepare the result for transmission to the final stage. The bulk of the processing is done by the function calls on lines 28-29. These are written in standard C.

4.1. Performance

In assessing the performance of a new programming model (in our case, *eSkel*), it is important to try to distinguish overheads introduced by the implementation mechanism from constraints which are inherent characteristics of the algorithm or underlying machine. To this end, the following programs were written:

- **Sequential**. This represents a "sensible" rendition of the algorithm. The images are processed in sequence by a loop and there is no unnecessary copying, as might be performed by a sequential emulation of the parallel algorithm. No attempts have been made to optimise lower level aspects (e.g. thinking about array access patterns to enhance cache utilisation), but this is equally true of our parallel versions.

- **Raw MPI**. A straightforward MPI implementation of the adapted algorithm, generalised to allow for cyclic distribution of image sets to a number of workers in the second stage (in other words, a hand-coded "deal", specialised to this application, and thereby omitting some of the data marshalling hidden inside the *eSkel* versions).

- **eSkel**. The "*Pipeline* and *Deal*" program described above.

```
1    spread_t spreads[STAGES+1]   = {SPGLOBAL, SPGLOBAL,SPGLOBAL,SPGLOBAL};
2    MPI_Datatype types[STAGES+1] = {MPI_INT, MPI_INT, MPI_INT, MPI_INT};
3    Imode_t imodes[STAGES]       = {EXPL, DEV, EXPL};
4    eSkel_molecule_t *(*stages[STAGES])(eSkel_molecule_t *) = {
5      (eSkel_molecule_t * (*)(eSkel_molecule_t *)) stage1,
6      (eSkel_molecule_t * (*)(eSkel_molecule_t *)) stage2,
7      (eSkel_molecule_t * (*)(eSkel_molecule_t *)) stage3
8    };
9
10   MPI_Init(&argc, &argv);
11   SkelLibInit();
12
13   MPI_Type_contiguous((ROWS+WINY)*COLS, MPI_INT, &MPI_int_array);
14   MPI_Type_commit(&MPI_int_array);
15   types[1]   = MPI_int_array;  types[2]   = MPI_int_array;
16
17   if (myrank()==0)                mystagenum = 0;
18   else if (myrank()==nprocs-1)    mystagenum = 2;
19   else                            mystagenum = 1;
20
21   Pipeline (STAGES,  imodes, stages, mystagenum, BUF, spreads, types,
22             NULL, 0, 0, NULL, 0, &outmul, 0, mycomm());
23
24   MPI_Finalize();
```

Figure 4. The main program

```
1  void stage2 (void) {
2    int outmul;
3
4    Deal (mycommsize(), IMPL, worker, myrank(), STRM,
5          NULL, 0, 0, SPGLOBAL, MPI_int_array,
6          NULL, 0, &outmul, SPGLOBAL, MPI_int_array, 0, mycomm());
7  }
8
9  eSkel_molecule_t * worker (eSkel_molecule_t *item) {
10   int *ref, *m1, *m2, i, j;
11
12   float curbesterr[ROWS*COLS];
13   int   curbestdisp[ROWS*COLS];
14   float diffimg[(ROWS+WINY)*COLS];
15
16
17   ref           =  &((int *)(item->data[0]))[0];
18   m1            =  &((int *)(item->data[0]))[(ROWS+WINY)*COLS];
19   m2            =  &((int *)(item->data[0]))[2*(ROWS+WINY)*COLS];
20
21   for (i=0;i<ROWS;i++)
22     for (j=0;j<COLS;j++) {
23         curbesterr[i*COLS+j]  = 9999999.0;
24         curbestdisp[i*COLS+j] = 0;
25     }
26
27   for (curdisp=0;curdisp<MAXDISP;curdisp++) {
28     gendiffimg(ref,m1,m2,diffimg,curdisp);
29     updatedispimg(diffimg,curbesterr,curbestdisp,curdisp);
30   }
31
32   item->len[0]  = 1;
33   item->data[0] = curbestdisp;
34   return item;
35 }
```

Figure 5. Code for stage 2.

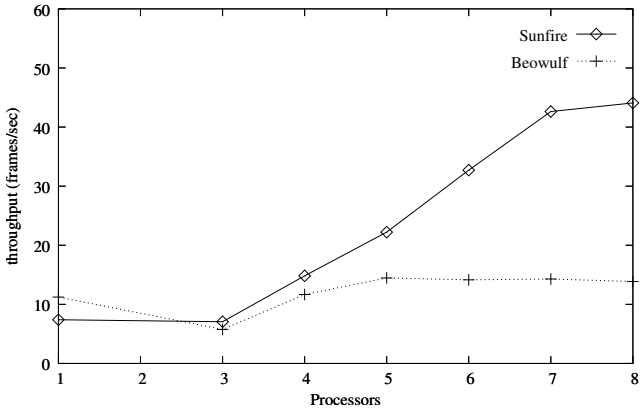


Figure 6. Performance of the raw MPI algorithm on the two machines.

Our experiments were conducted on two platforms:

- A 64 node (1.8 GHz Intel Pentium 4 processors) Beowulf cluster networked with a 100Mb ethernet, using the Los Alamos Message Passing Interface LA-MPI.
- A 52 node (0.9 GHz Ultrasparc III processors) Sunfire E15k SMP with the native Sun implementation of MPI.

All runs were repeated six times, with results shown below being averages. None of the runs deviated from the average to any interesting extent. All experiments involved the processing of 20 images, each of 240×256 pixels. We measure performance in terms of throughput, as described above.

4.2. Performance of the underlying algorithm

Before assessing *eSkel*, it is important to understand the capabilities of the parallelised multiple baseline stereo algorithm itself. Figure 6 compares the performance of the sequential and raw MPI programs on each of the platforms. The sequential performance (data points for one processor) is a little better on the Beowulf, presumably reflecting the greater speed of its processors. There are no data points for two processors because our chosen parallelisation requires at least three processors. For four and more processors, we see the effect of adding extra workers to the second stage: the data points for p processors indicate the use of a directly programmed "deal" with $p - 2$ workers. We notice that performance on the Beowulf is disappointing, with the parallel versions only eventually marginally beating the sequential one, and with the curve flattening very quickly. This suggests that the relatively low computation to communication ratio inherent in the algorithm is too much for this machine to cope with. In contrast, the situation on the SunFire is much more promising, with parallelisation apparently worthwhile, (flattening at about eight processors).

4.3. Performance of *eSkel*

We can now investigate the performance of *eSkel*. Figure 7 compares the throughput of the raw MPI and corresponding *eSkel* programs on the Beowulf. The first data point is for three processors because, as noted above, this is the smallest natural number of processors for a parallel version. We observe that the two programs perform quite similarly, with a performance loss of around 8% for *eSkel*. Figure 8 illustrates a similar comparison for the Sunfire. The overall pattern is similar, with a performance gap which is slightly wider (around 10-13%). We are encouraged by these results, particularly since we have already observed that the application is inherently communications-heavy,

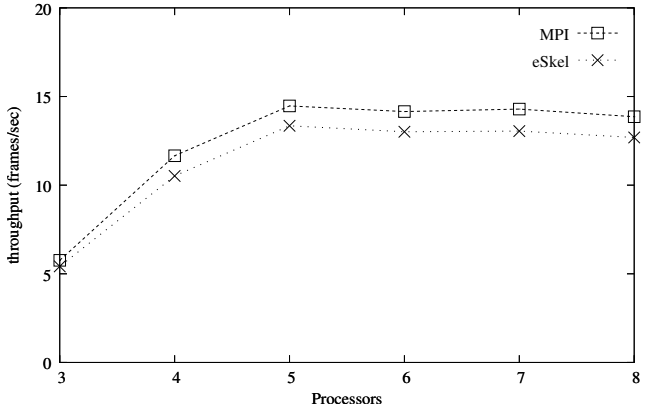


Figure 7. Performance of *eSkel* and MPI on the Beowulf.

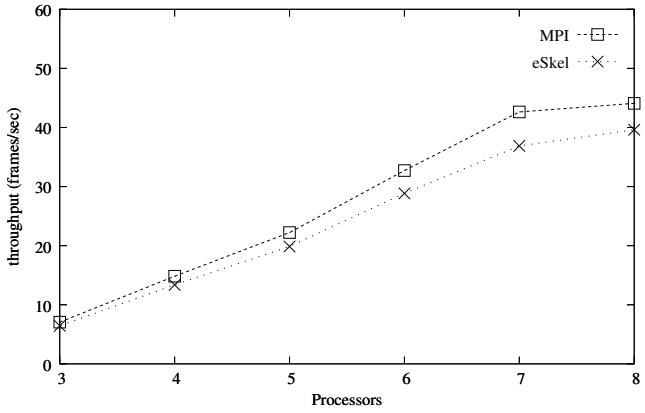


Figure 8. Performance of *eSkel* and MPI on the Sunfire.

and so can be expected to highlight any weaknesses in the underlying system. We additionally note that the current implementation of *eSkel* is a proof-of-concept prototype, with considerable scope for optimisation, and expect that the performance gap highlighted here can be significantly closed.

The observant reader may wonder why the performance gap is more noticeable for the Sunfire, when intuition might suggest that this machine, with its relatively superior communications technology, might be more forgiving of communications profligacy. One credible hypothesis might suggest that this is related to the relative merits of the implementations of MPI on the two machines: *eSkel* makes heavy use of collective communications (particularly `MPI_Scatterv` and `MPI_Gatherv`). If these operations were less effectively implemented (with respect to simple sends and receives) on the SunFire than on the Beowulf, then we would expect the observed effect. However, simple experiments with the operations were unable to produce any convincing evidence. Another hypothesis would suggest that there may be a cache-related effect at work. Remembering that all "communication" in the Sunfire is underpinned by shared memory, might it be the case that the sends and receives of the raw MPI versions have better cache side-effects than the scatters and gathers of the *eSkel* implementations?

5. Conclusions

We conclude with some observations motivated by the assessment criteria laid out in section 2.

1. **Conceptual disruption**. The "skeletons as collective operations" approach seemed to fit the algorithm neatly. The pointer twiddling and casting to interface to incoming and outgoing data was messy. While this is partly inherited from MPI (the raw program needed similar code to allow the arrays to be transmitted efficiently), the molecule concept added a level of indirection.

2. **Ad-hoc parallelism**. The application itself is too regular to need this, but the performance monitoring code (omitted from the extracts here) exploited this facility.

3. **Flexible skeletons**. It was helpful to be able to program a pipeline in which the data stream was generated directly by the first stage.

4. **Performance**. As noted above, this seems to be at least encouraging.

6. Acknowledgements

The authors are supported by the Enhance project ("Enhancing the Performance Predictability of Grid Applications with Patterns and Process Algebras") funded by the Engineering and Physical Sciences Research Council under grant number GR/S21717/01.

References

[1] A. Benoit and M. Cole. *eSkel's web page*. `http://homepages.inf.ed.ac.uk/mic/eSkel`.

[2] A. Benoit and M. Cole. Two Fundamental Concepts in Skeletal Parallel Programming. In V. Sunderam, D. van Albada, P. Sloot and J. Dongarra (eds), *International Conference on Computational Science (ICCS), Part II*, LNCS 3515(764–771), Springer, 2005.

[3] A. Benoit, M. Cole, S. Gilmore, and J. Hillston. Flexible Skeletal Programming with eSkel. *To appear in Proceedings of EuroPar 2005*, LNCS, Springer, 2005.

[4] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press & Pitman, ISBN 0-262-53086-4, 1989.

[5] M. Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30(3):389–406, 2004.

[6] M. Okutomi and T. Kanade. A Multiple-Baseline Stereo. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(4):353–363, April 1993.

[7] S. Pelagatti. *Structured Development of Parallel Programs*. Taylor & Francis, ISBN 0-7484-0759-6, London, 1998.

[8] F.A. Rabhi and S. Gorlatch, editors. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer Verlag, ISBN 1-85233-506-8, 2003.

[9] J. Subhlok, D. O'Hallaron, T. Gross, P. Dinda, and J. Webb. Communication and memory requirements as the basis for mapping task and data parallel programs. In *Proceedings of Supercomputing '94*, pages 330–339, Washington, DC, November 1994.

[10] J.A. Webb. Latency and Bandwidth Considerations in Parallel Robotics Image Processing. In *Supercomputing'93*, pages 230–239, Portland, OR, November 1993.

A Java/Jini framework supporting stream parallel computations

M. Danelutto[a] & P. Dazzi [b c]

[a]Dept. Computer Science – University of Pisa – Italy

[b]ISTI/CNR – Pisa, Italy

[c]IMT – Lucca Institute for Advanced Studies – Italy

JJPF (the Java/Jini Parallel Framework) is a framework that can run stream parallel applications on several parallel-distributed architectures. JJPF is a distributed execution server, actually. It uses JINI to recruit the computational resources needed to compute parallel applications. Parallel applications can be run on JJPF provided they exploit parallelism accordingly to an arbitrary nesting of task farm and pipeline skeletons/patterns. JJPF achieves almost perfect, fully automatic load balancing in the execution of such kind of applications. It also transparently handles any number of node and network faults. Scalability and efficiency results are shown on workstation networks, both with a synthetic (embarrassingly parallel) image processing application and with a real (not embarrassingly parallel) page ranking application.

1. Introduction

It is generally assessed that real parallel applications usually exploit parallelism according to a limited, well-known set of patterns (or skeletons) [9,21,20,7]. With the advent of grids [13,14] and large cluster architectures [25] some of the parallelism exploitation patterns originally proposed in the skeleton framework have been extensively used to implement high performance, parallel grid applications. Indeed, very often parallel applications are programmed exploiting *by hand* typical grid middleware or operating system/distributed framework mechanisms without even stating they owe to the algorithmic skeleton or parallel design patterns most of the techniques use to exploit parallelism. An example of parallelism exploitation pattern that is very often used both in grids and in more traditional distributed frameworks is the task farm one. In a task farm, a set, or a stream, of independent tasks are computed to obtain a set of results. A single program or function is used to compute all the results out of the input tasks. Such parallelism exploitation pattern is also referred to as *embarrassingly parallel computations* [27]. All the parameter sweeping applications, that is those applications that "try" input data sets to find out the best one with respect to some measure function, can be easily programmed exploiting parallelism according to the task farm pattern. Also, most of the grid applications that can be programmed using tools such as Condor [11] (basically a batch job scheduler) can be programmed as task farm instances. Another well-known and used parallelism exploitation pattern is the pipeline one. In a pipeline a set of input tasks are processed by a set of stages. Each stage just computes a result out of the result provided by the previous stage and delivers result to the immediately following stage. Task farm and pipeline parallelism exploitation patterns are often referred to as stream parallel (or task parallel) patterns/skeletons [23,20]. We already demonstrated that arbitrary compositions of pipeline and task farm patterns can be efficiently implemented, with respect to service time, using their normal form, that is transforming the original skeleton tree/composition into a program that is basically a task farm with sequential workers [3].

[0]This work has been partially supported by Italian national project no. RBNE01KNFP *GRID.it* and No. 02.00640.ST97 and by the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

Overall, this allows us to conclude that if we succeed providing a distributed environment efficiently exploiting task parallel computations, this will be very useful to implement different applications in really different applicative and hardware contexts. Our group already published several works related to the implementation of such kind of programming environments [7,4,12,8] and it is currently involved in a large national project (the Italian FIRB project GRID.it [17]) aimed at designing and implementing a prototype, high performance, structured programming environment [26,2,1]. In this work, we discuss a parallel programming framework (JJPF) built on top of plain Java/Jini that can run stream parallel applications on several parallel/distributed architectures ranging from tightly coupled workstation clusters to generic workstation networks and grids. The framework directly inherits from Lithium and `muskel`, two skeleton based programming environments we previously developed at our Department [4,12]. Both Lithium and `muskel` exploit plain RMI Java technology to distribute computations across nodes, and rely on NFS (the network file system) to distribute user code to the remote processing elements. JJPF, instead, is fully implemented on top of Jini/Java and relies on either Jini/Jeri class loaders or on a brand new, hybrid class loader package, to distribute code across the remote processing nodes involved in stream parallel application computation. JJPF exploits the stream parallel structure of the application in such a way that several distinct goals can be achieved: a) *load balancing* across the computing elements participating in the computation b) *automatic discovering and recruiting* of processing elements available to participate to the computation of stream parallel applications exploiting standard Jini mechanisms c) *automatic substitution of faulty processing elements* by fresh ones (if any). Therefore the stream parallel applications computations resist to both node and network faults. The programmer does not need to add a single line of code in his application to deal with faulty nodes/network, nor he has to take any other kind of action to get advantage of this feature. JJPF has been tested using both synthetic and real applications, on both production workstation networks and on a blade cluster, with very nice and encouraging results, as described in Section 3.

2. JJPF

JJPF has been designed to provide programmers with a user-friendly environment supporting the efficient execution of stream parallel applications on a network of workstations, exploiting plain, state of the art, Java technology. Overall JJPF provides a distributed server providing a stream parallel application computation service. Programmers must write their applications in such a way they just exploit an arbitrary composition of task farm and pipeline patterns. Task farm only applications are directly executed by the distributed server, while applications exploiting composition of task farm and pipeline patterns are first processed, in a completely automatic way, to get their normal form [3] and then their normal form is submitted to the distributed server for execution. Using JJPF, programmers can express a parallel computation exploiting the task farm pattern simply using the following code:

```
BasicClient cm = new BasicClient(program,null,input,output); cm.compute();
```

provided that `input` (`output`) is `Collection`of input (output) tasks and `program` is an array hosting the worker code of the farm. The worker code is a `Class` object relative to the user worker code. Such code must implement a `ProcessoIf` interface. The interface requires the presence of methods to provide the input task data (`void setData(Object task)`), to retrieve the result data (`Object getData()`) and to compute result out of task data (`void run()`). This single line of code actually defines the parallel computation to be executed, starts its execution and terminates when the parallel execution is actually terminated. JJPF basic architecture uses two components: clients, that is the user programs, and service, that is distributed server instances that

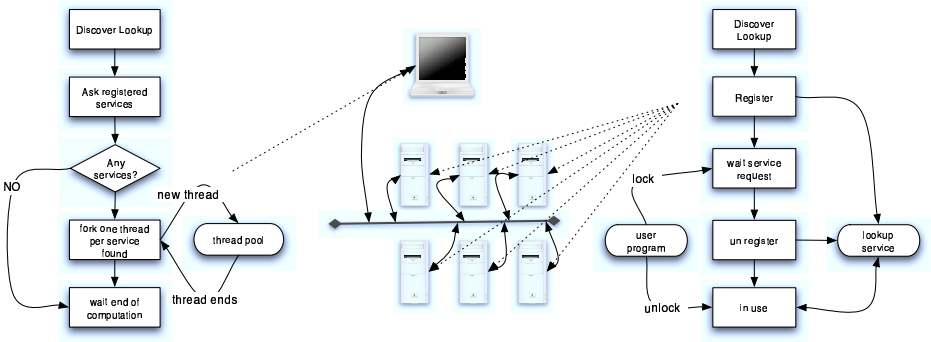


Figure 1. Simplified state diagram for the generic JJPF *client* (left) and *service* (right)

actually compute results out of input task data to execute client programs. Figure 1 sketches the structure of these two components. The client component basically recruits available services and forks a control thread for each one of them. The control thread, in turn, fetches un-computed task items from the task repository, delivers them to the remote service and retrieves the computed results, storing them to the result repository. Service recruiting is performed exploiting JINI. A lookup service is found first, using standard JINI API, then it is queried for available services. Each service descriptor obtained from lookup is passed to a distinct control thread. On the other hand, the service registers to the JINI lookup, and waits for incoming client calls. Once a call is received, it assumes to be recruited by that client, un-registers from the lookup and starts serving task computation requests from the client. This means that clients actually *lock* the services recruited to their exclusive usage. Therefore, in order to use JJPF on a workstation network, the following steps have to be performed: a) JINI has to be installed and configured (this has to be done once and for all, of course), b) JJPF services has to be started at the machines that will eventually be used to run the JJPF distributed server (this also is to be done once and for all), and c) a JJPF client such as the one sketched above has to be prepared, compiled and run on the user workstation. Nothing else is needed. The key concept in JJPF is that service discovery is automatically performed in the client run time support. Not a single line of code dealing with service discovery or recruiting is to be provided by application programmers. Both these mechanisms rely on the JINI technology. This means that all the power of this technology is exploited but also that some limitations of the technology are inherited. In particular, we worried about the fact that JINI discovery mechanisms cannot pass through firewalls, therefore impairing JJPF usability in grid or in large distributed architecture contexts. Indeed, the JINI technology is perfectly suitable to run on workstation clusters within local area networks. JJPF uses two distinct mechanisms to recruit services to clients. It directly requires to the Lookup Service the Service Ids of the available services, i.e. of the nodes currently running the JJPF generic service object, but it also registers to the Lookup Service *observer* objects that will eventually advise the client of new services becoming available, in such a way they can be recruited. When implementing JJPF we had to face the problem of making available user code (the one computing a result out of the single task) to the remote services. JJPF achieves automatic load balancing among the recruited services, due to the scheduling adopted in the control threads managing the remote services. Each control thread fetches tasks to be delivered to the remote nodes from a centralized, synchronized task repository. JJPF also automatically handles faults in service nodes. That is, it takes care of the tasks assigned to a service node in such a way that in case the node does not respond any more they

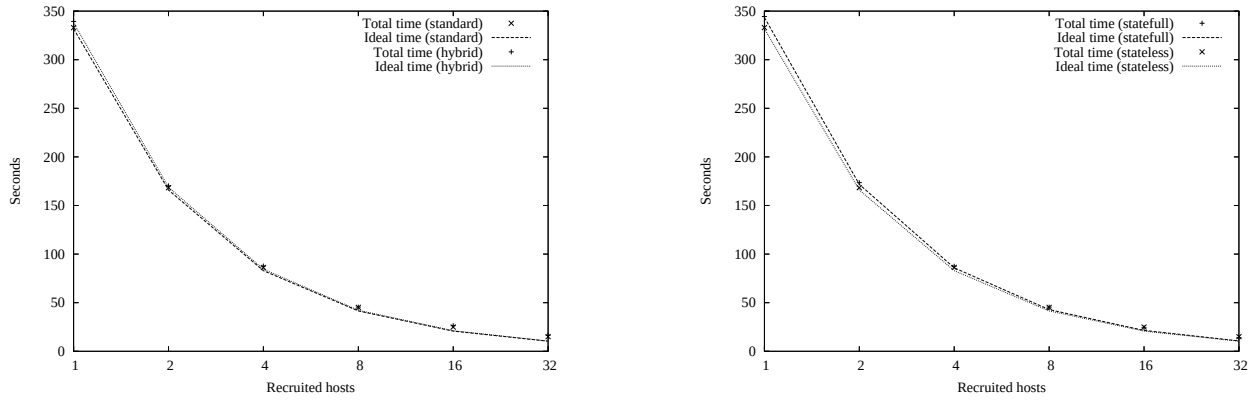


Figure 2. Scalability of JJPF: scalability on a production workstation network, image processing application with standard class loader and with hybrid class loader (left); same application application with (stateful) and without (stateless) access to a shared variable (right)

can be rescheduled to other service nodes, possibly recruited on the fly after realizing the service node fault. This is only possible because of the kind of parallel applications we are taking into account and that are supported in JJPF, that is stream parallel computations. In this case, there are natural *descheduling points* that can be chosen to restart the computation of one of the input tasks, in case of failure of a service node. A trivial one is the start of the computation of the task. Provided that a copy of the task data is kept on the client side (in the control thread, possibly), the task can be rescheduled as soon as the control thread understands that the corresponding service node is death/non responding. This is the choice we actually implemented in JJPF, inheriting the design from `muskel` [12].

3. Experiments

In order to test JJPF features and scalability, we used two kind of applications. Most of the simple scalability measures have actually been performed using a synthetic image processing application. The application just filtered all the images appearing of an image set, applying a sort of blur image filter. This synthetic application mimics real applications that are used, as an example, to pre-process images coming from satellites, telescopes, etc. just before storing them to disks for further, real processing. The image filtering application is actually an embarrassingly parallel application, perfectly matching the task farm pattern. After verifying the scalability and efficiency results of JJPF with the synthetic application, we decided to use a complete application. As there are several people in our department working on web applications, we thought to exploit the available knowledge to develop a page ranking application. The goal was to have a real application at hand that can be used to confirm the scalability and efficiency results achieved with the synthetic case study. The page rank application we developed works with an approximate algorithm. In general, the rank vector x is iteratively computed in such a way that $x^{(k)} = Ax^{k-1}$ until $|| \ x^{(k)} - x^{k-1} \ ||> \epsilon$ [16]. In the approximate algorithm, a pre-processing phase distributes the vector x and the matrix A across a set of services, in such a way that each service can compute a part of the new intermediate rank vector. Once this has been computed, the result is exchanged with the other services in such a way a new iteration (group of iterations) can be computed. The approximate algorithm does not compute the exact page ranking, of course, but the approximation introduced does not impairs the

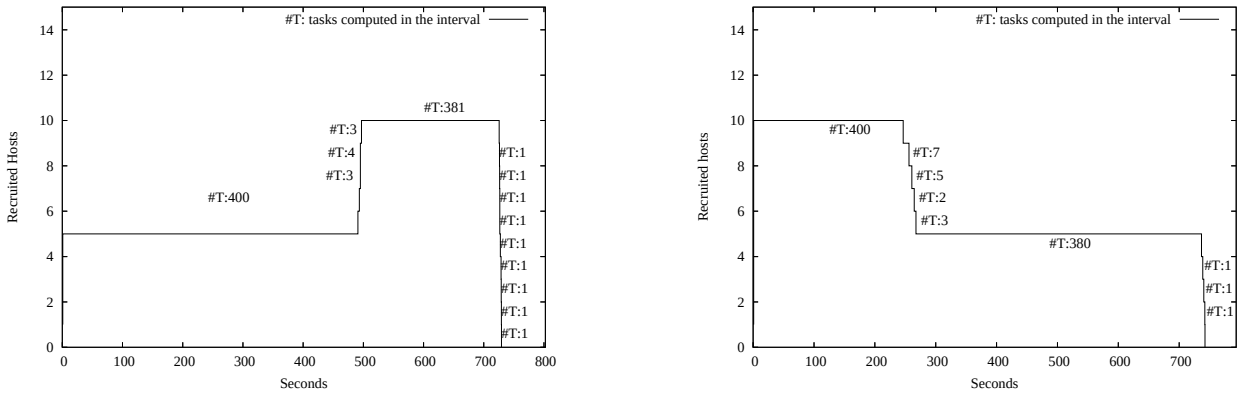


Figure 3. Effect of discovering/recruiting new resources to (left) or dismissing (faulty) resources from (right) the current computation

effectiveness of the algorithm itself. Most page ranking algorithms refer to similar approximation techniques [24,22]. Using these two applications, we run a set of experiments using JJPF to test the feasibility and the efficiency of our approach. We used two distinct kind of distributed architectures: a network of "production" Linux workstations and a cluster of blade PCs, also operated by Linux. The production workstation network was a highly dynamic environment. These workstations are dual boot (Debian Linux and Windows XP), Pentium IV class machines used by the students for their class work. They are often rebooted to switch the operating system and therefore you cannot assume that they stay constantly up and running. Moreover, the users (the students) usually run a variety of tasks on these machines, ranging from WEB browsers to huge compilation and execution tests. The blade machines, on the other side, are based on RLX Pentium III blades, with 3 fast Ethernet networks interconnecting the blades arranged in a single chassis. We have total control on the blade cluster and therefore we could run the tests on "dedicated" nodes. First of all, we tested the scalability of our distributed computation server. We used the synthetic image filtering application to process a stream of input images. The results are shown in Figure 2. In the left part of the Figure, the completion times achieved when the standard class loader mechanism was exploited are shown. In the right part of the Figure, the completion time achieved using our hybrid class loader mechanism is shown. In both cases, we plot the ideal completion time (that is the time spend to compute sequentially all the filtered images divided by the processing elements actually used), the measured completion time and the time actually spent in the computation of the filtered images. Scalability is actually achieved. In all cases, the efficiency was above 90%. Then we measured the efficiency of the recruiting, dismissing mechanisms of JJPF. Therefore we set up two experiments. In the first one, a number of workstations are initially recruited, and further workstations are recruited after that half of the tasks (filtered images) have been already been computed (see Figure 3 left). In the second one, after initially recruiting a number of workstations, half of them are lost (verified faulty) after the computation of half of the tasks (filtered images) (see Figure 3 left). In both cases, the time spent in computing the half tasks with the half workers available took the double of the time taken to compute the other half of the tasks, as expected. The `#T` numbers in the plots, refer to the number of tasks computed in that segment of the plot line. As an example, in the left plot of Figure 3 the `#T:400` indicates that using the initially recruited machines, we computed 400 tasks (filtered images) before actually starting recruiting other machines. When all the additional machines where recruited and just before starting dismissing machines, we computed a further `#T:381` tasks. In

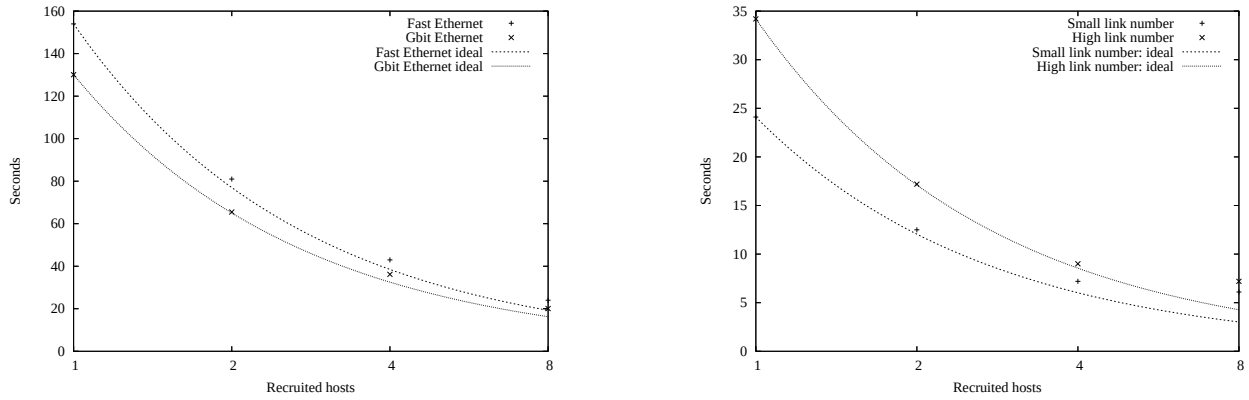


Figure 4. Scalability of JJPF (page rank application): Fast Ethernet vs. Gbit Ethernet (left) Small vs. high link number per page (right)

these experiments, new computing nodes/services are made available for recruitment by running the JJPF run time on new machines and faulty nodes are emulated either by stopping the JJPF support or by unplugging network cables from the switch. Both the experiments whose results are plotted in Figures 2 and 3 have been performed using the network of production workstations. Eventually, we run an experiment with a real application code, the page rank algorithm described above. This experiment has been performed using the blade cluster. We achieved comfortable results, although the scalability measured is not equal to the one achieved using the synthetic, embarrassingly parallel application. The point here is that we need to exchange data among the workers participating in the computation to take care of the approximation algorithm used in the page rank. The right part of the Figure 4, plots the completion times of the page rank application run on a set of 400K pages (therefore a fairly small set of pages) with each page holding a reasonable, but fairly poor number of links to other pages, as well as the completion times achieved using another set with the same number of pages but with pages that hold a quite larger number of links. This was to point out the effect of computation grain on the scalability. In the former case, we compute less before actually starting exchanging the data needed to compute the page rank approximation. In the latter, we compute more. Therefore we pay a smaller (percentage) overhead in the latter case and scalability turns out to be better. In the left part of Figure 4, we point out the differences achieved when using a Gbit Ethernet interconnection between blades instead of a plain Fast Ethernet, 100Mbit interconnection. The network shift improved the completion times, although it did not change significantly the shapes of the completion time curves. Overall, we can state that JJPF demonstrated the expected scalability results as well as its ability too dynamically handle new computational resources, when available, and to safely dismiss nodes (without actually loosing any kind of data), in case they stop working.

4. Related work

Our previous full Java, structured, parallel programming environment `muskel` already provides automatic discovery of computational resource in the context of a distributed workstation network. `muskel` was based on plain RMI Java technology, however and the discovery was simply implemented using multicast datagrams and proper discovery threads. The `muskel` environment also introduces the concept of *application manager* that binds computational resource discovery with autonomic application control in such a way that optimal resource allocation can be dynamically

maintained upon specification by the user of a performance contract to be satisfied [12]. Several other groups proposed or currently propose environments supporting stream parallel computations on workstation networks and clusters. Among the others, we mention Cole's *eSkel* library running on top of MPI [10], Kuchen's C++/MPI skeleton library [20] and CO_2P_2S from the University of Alberta [21]. The former two environments are libraries designed according to the Cole algorithmic skeleton concept. The latter is based on parallel design patterns. None of them allows, at the moment, automatic discovery of computational resources, nor provides fault tolerance features such as those provided by JJPF. The group of Françoise André is currently trying to address the problem of dynamically varying the computational resources assigned to the execution of an SPMD program [6]. This is not actually the same problem we addressed with JJPF, but the techniques used to devise the exact number of resources to be recruited to compute a parallel program are interesting and can be reused in JJPF framework to recruit the right number of service nodes among those available. Our group is also introducing dynamicity handling techniques in the ASSIST environment developed within the GRID.it project [5]. Such techniques are partially derived from the muskel/JJPF experience. The kind of task parallel computations natively supported by JJPF is very close to the one supported by Condor. Condor is a "specialized workload management system for compute-intensive jobs" [11] and "like other full-featured batch systems, it provides a job queuing mechanism, scheduling policy, priority schema, resource monitoring and resource management". However, Condor is a actually a batch system, that is it is not a programming environment, nor it is able to provide (as JJPF does through skeletons and normal form) support for other, different parallelism exploitation patterns/skeletons. Several papers are related to PageRank Algorithm, Haveliwala [18] explores memory-efficient computation, in [19]. Kamvar et al. discuss some methods for accelerating PageRank calculation and in [15] Gleich, Zhukov and Berkhin demonstrate that linear system iterations converge faster than the simple power method and are less sensitive to the changes in teleportation. Rungsawang and Manaskasemsak in [24] e [22] evaluate the performance supplied by an approximated PageRank computation on a Cluster of Workstation using a low-level peer-to-peer MPI implementation.

5. Conclusions

We described JJPF a new distributed server supporting the execution of stream parallel application on workstation networks. The framework exploits plain Java technology, using JINI to address resource discovery. JJPF supports the execution of stream parallel computations using a set of remote service nodes, that is, nodes that basically provide a sort Java interpreter capable of computing generic, user-defined tasks implementing a known interface. Service nodes are discovered and recruited automatically to support user applications. Fault tolerance features have been included in the framework such that the execution of a parallel program can transparently resist to node or network faults. Load balancing is guaranteed across the recruited computational resources, even in case of resources with fairly different computing capabilities. To our knowledge, these features are not present in other distributed parallel programming environments.

References

[1] M. Aldinucci, S. Campa, P. Ciullo, M. Coppola, S. Magini, P. Pesciullesi, L. Potiti, R. Ravazzoloand M. Torquati, M. Vanneschi, and C. Zoccolo. The Implementation of ASSIST, an Environment for Parallel and Distributed Programming. In *Proc. of Intl. Conference EuroPar2003: Parallel and Distri buted Computing*, number 2790 in LNCS. Springer, 2003.

[2] M. Aldinucci, S. Campa, M. Coppola, M. Danelutto, D. Laforenza, D. Puppin, L. Scarponi, M. Vanneschi, and C. Zoccolo. Components for High-Performance Grid Programming in GRID.it. In *Component modes and systems for Grid applications*, CoreGRID. Springer, 2005.

[3] M. Aldinucci and M. Danelutto. Stream parallel skeleton optimisations. In *Proc. of the IASTED International Conference Parallel and Distributed Computing an d Systems*, pages 955–962. IASTED/ACTA Press, November 1999. Boston, USA.

[4] M. Aldinucci, M. Danelutto, and P. Teti. An advanced environment supporting structured parallel programming in Java. *Future Generation Computer Systems*, 19(5):611–626, 2003. Elsevier Science.

[5] M. Aldinucci, A. Petrocelli, E. Pistoletti, M. Torquati, M. Vanneschi, L. Veraldi, and C. Zoccolo. Dynamic reconfiguration of grid-aware applications in ASSIST. In *11th Intl Euro-Par 2005: Parallel and Distributed Computing*, LNCS. Springer Verlag, 2005.

[6] F. André, J. Buisson, and J.L. Pazat. Dynamic adaptation of Parallel Codes: Toward Self-Adaptable Components. In *Component modes and systems for Grid applications*, CoreGRID. Springer, 2005.

[7] B. Bacci, M. Danelutto, S. Pelagatti, and M. Vanneschi. SkIE: a heterogeneous environment for HPC applications. *Parallel Computing*, 25:1827–1852, Dec. 1999.

[8] R. Baraglia, M. Danelutto, D. Laforenza, S. Orlando, P. Palmerini, R. Perego, P. Pesciullesi, and M. Vanneschi. AssistConf: A Grid Configuration Tool for the ASSIST Parallel Programming Environment. In *11th Euromicro Conf. on Parallel, Distributed and Network- Based Processing*, pp 193–200. IEEE, 2003.

[9] M. Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30(3):389–406, 2004.

[10] M. Cole and A. Benoit. The eSkel home page, 2005. http://homepages.inf.ed.ac.uk/abenoit1/eSkel/.

[11] Condor community. The CONDOR home page, 2005. http://www.cs.wisc.edu/condor/.

[12] M. Danelutto. QoS in parallel programming through application managers. In *Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network -based processing*. IEEE, 2005. Lugano.

[13] I. Foster and C. Kesselman (Editors). *The Grid 2 Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, December 2003.

[14] Ggf community. The Global Grid Forum home page, 2005. http://www.gridforum.org.

[15] D. Gleich, L. Zhukov, and P. Berkhin. Fast Parallel PageRank: A Linear System Approach. Technical report, Yahoo research lab, 2004.

[16] Google community. The Google home page, 2005. http://www.google.com/technology/.

[17] Grid.it community . The GRID.it home page, 2005. http://www.grid.it.

[18] Taher Haveliwala. Efficient Computation of PageRank. TR 1999-31, Stanford Univ., USA, 1999.

[19] S. Kamvar, T. Haveliwala, C. Manning, and G. Golub. Extrapolation methods for accelerating PageRank computations, Proceedings of the Twelfth Int'l WWW Conference, 2003.

[20] H. Kuchen. A Skeleton Library. In *Euro-Par 2002, Parallel Processing*, number 2400 in LNCS, pages 620–629. "Springer" Verlag, August 2002.

[21] S. MacDonald, J. Anvik, S. Bromling, J. Scaheffer, D. Szafron, and K. Tan. From patterns to frameworks to parallel programs. *Parallel Computing*, 28(12), 2002.

[22] Bundit Manaskasemsak and Arnon Rungsawang. Parallel PageRank Computation on a Gigabit PC Cluster. In *AINA (1)*, pages 273–277, 2004.

[23] S. Pelagatti. *Structured Development of Parallel Programs*. Taylor & Francis, 1998.

[24] Arnon Rungsawang and Bundit Manaskasemsak. PageRank Computation Using PC Cluster. In *PVM/MPI*, pages 152–159, 2003.

[25] Univ. of Tennesse, Mannheim and NERSC/LBNL. The Top500 home page. www.top500.org, 2005.

[26] M. Vanneschi. The Programming Model of ASSIST, an Environment for Parallel and Distributed Portable Applications. *Parallel Computing*, 12, December 2002.

[27] B. Wilkinson and M. Allen. *Parallel Programming: Techniques and Applications Using N etworked Workstations and Parallel Computers*. Prentice Hall, 1999.

Architecture

Massively parallel MIMD architecture achieves high performance in a spam filter

O.R. Birkeland[a], M. Nedland[a], O. Snøve Jr.[a]

[a]Interagon AS, Medisinsk teknisk senter, NO-7489 Trondheim, Norway

Supercomputer manufacturing is usually a race for floating point operations, and most therefore opt for a design that allows for the highest possible clock frequency. Many modern applications are, however, limited not only by the number of operations that can be performed on the data, but by the available memory bandwidth. We review the main features of a MISD architecture that we have introduced earlier, and show how a system based on these chips is able to scale with respect to query and data volume in an email spam filtering application. We compare the results of a minimal solution using our technology with the performance of two popular implementations of deterministic and nondeterministic automatons, and show how both fail to scale well with neither query nor data volumes.

1. Introduction

Mainstream parallel computer systems have diverse design targets. Projects like BlueGene/L aim for the highest performance on numerical calculations [6], whereas designs like Green Destiny target efficiency, reliability and availability instead [2]. A common characteristic of most designs is the dominating supercomputer requirement for floating point operations, but several applications, for instance within bioinformatics and network traffic monitoring, are limited by the available memory bandwidth rather than the raw floating point compute power [8,1]. Brute force data mining, that is linear searching through raw data, is one such application.

Fine grain parallelism have been proposed by others, both as minute compute nodes [4] or integration of memory and processing circuitry [8,5]. In our work, we propose an architecture that takes advantage of one of the main assets of synchronous memory technology, which is high sustainable bandwidth during linear access. We review a MIMD architecture for pattern matching with regular expression-type queries, with high performance, high density and low infrastructure requirements. By tailoring the compute logic for the given problem, the system's size and complexity can be reduced to a minimum. Focusing on non-computational applications, we are able to build systems with higher density and more operations per second than traditional clusters and supercomputers.

One application that requires scalability with respect to both query and data volume is email spam filtering. Spam is prevented with a large number of approaches, but most involve (among others) using regular expressions to find common spam patterns. The number of required spam patterns are increasing, as well as the volume of email, resulting in an increasing demand for compute power. Some of the common regular expression evaluators like DFA scanners have severe scalability issues. We propose a scalable architecture for evaluation queries such as regular expressions, that achieves high performance in a compact system by use of fine grain parallel processing.

2. System architecture

We concentrate on the design considerations for this architecture, including general purpose versus special purpose hardware; fine grain versus coarse grain parallelism; compute density; cost of ownership; and requirements for development resources.

Our project started with the observation that an index-based algorithm requires that the keywords are known beforehand, and the realization that an extremely rapid brute force linear search eliminates the need for an indexing step. A continuous linear search also implies that searching becomes independent of the queries's complexity. Since the execution time for such a search is $O(n)$, it becomes important to use the peak memory bandwidth, and have as many independent memory channels as possible.

We developed a fine-grain parallel architecture called the pattern matching chip (PMC) [3], where each node can be as small as a single ASIC and a single memory device. Each PMC contains 1,024 individual processing elements operating on a shared data stream. We constructed a MIMD system with PCI cards, each containing 16 PMC chips, accelerating a standard PC to perform 10^{13} symbol comparisons per second. Each chip has a dedicated memory bank; thus, the memory volume and bandwidth scale at the same rate as the number of processing elements (PEs).

16 or more PEs are used for each query, depending on query size. This allows for up to 64 simultaneous queries per chip. Additional queries can be stored in the local memory, requiring 16 kB for each configuration for all 1024 PEs. Thus several thousand queries can be resident for each PMC, and be used with minimal system involvement, only to specify which queries to use. In spam filtering, the email data can be uploaded to local memory once, and then interactively processed by several sets of queries. If the data rate is lower than the scan rate, the difference can be reclaimed in the form of an increased number of queries. For example, one chip could scan 64 queries at 100 MB/s data rate, or 6400 queries at 1 MB/s data rate. Query configurations, as well as data, can be loaded to local SDRAM, even during searches, without any performance penalties. The performance is linear with respect to the number of queries, the length of the queries, and the data rate. It is also linearly scalable with more PMC chips.

The processing speed is limited by the memory bandwidth in each node. Consequently, the PMC is tailored to run at the memory data rate. As we did not aim for the highest processing clock frequency, a low power design was possible, and that proves essential when small building blocks are used to build dense systems. As all memory accesses take place as linear scans, the required power for searching though the data is minimized as well. Any non-predictable access pattern involves protocol overhead, for example in switching pages within the memory chip, during which a lot of time and energy are consumed.

The same methodology is applied to the configuration of the chip. The entire configuration is read once into the chip and stored in configuration registers. Thus, there are no instruction memory accesses during searches. As a side effect, the full memory bandwidth is available for data transfer.

The large number of small PEs on each chip implies no local hot spot. The same applies when several chips are used on the PCI card, without a requirement for active cooling. The density that can be achieved in the system is thus governed by the transistor density on silicon (PMC and SDRAM), rather than the power consumption density.

Another feat of this design is that smaller cores also imply less engineering. As we target applications where data can easily be distributed for processing, no elaborate interconnection scheme is required. The only custom part in the design is the PMC, and the remaining components are sourced from standard technologies like SDRAM and PCI. Correspondingly, the use of a small, repetitive, orthogonal core element also makes the compiler design simpler. Such compilation also involves minimal optimization steps, and executes very fast. DFA scanners like `lex` could achieve high scan rates, but with an unpredictable compile time.

The architectural concepts have been proven in a five PC cluster, achieving $5 \cdot 10^{13}$ operations per second with near-linear scalability. Performance is achieved by massively distributed compute

resources, which translates to 500,000 processing elements in the aforementioned cluster. Furthermore, our evaluation system has a total of 60 GB of memory, with an aggregated bandwidth of 48 GB per second.

We will compare the performance of the PMC with two popular software algorithms in a email spam filtering application. Our reason for choosing this application as our example, is that we are able to show how we can tailor the performance to yield the appropriate ratio of search speed to query throughput.

3. Problem definition

Spam—unsolicited commercial emails or unsolicited bulk emails—has become a tremendous problem for email users, and is increasing. Users around the world receive billions of spam messages each day and are indirectly forced to bear the cost of delivery and storage of these unwanted emails. A number of ways exist to deal with the problem—they range from simple blacklists of bad senders, words, or IP addresses to complex automated filtering approaches. One of these popular filters is SpamAssassin, a program that uses hundreds of weighted rules to obtain a score for each email header (see `http://spamassassin.apache.org/` for additional details). Patterns are constructed to match popular words used by spammers and a genetic algorithm optimizes the weights so that the combined predictor achieves optimal performance on a manually curated training set. Users must determine the actions that should be taken when an email receives a score that puts a spam label on the message according to the user's predefined thresholds.

SpamAssasin, and similar automatic filters, ensures that a message cannot be deemed spam based on a single rule, and conversely, that a message cannot bypass the filter just by avoiding a single rule. Such filters must evaluate many complex rules per message, which may not be a problem if you have a limited number of users on your system. Centralized hubs, such as those operated by internet service providers, however, must process a substantial query volume with high bandwidth message streams. This almost invariably leads to performance problems because most solutions does not scale well with both query and message volume.

Spam rules of an automatic filter are examples of queries that can be efficiently processed by our PMC. A trigger word such as `viagra` can be written in many ways, including for instance `v|agra`, `vìagrá`, `v*i*a*g*r*a`, and so on. The name of Pfizer's popular treatment for erectile dysfunction can actually be written in well above hundred different ways that must be dealt with by spam filters. We wanted to investigate the performance of widely used regular expression algorithms and compare their performance against that of our PMC-accelerated workstation.

We choose to compare our PMC system with efficient implementations of deterministic and nondeterministic automatons. Deterministic automatons are expected to be fast at the expense of large compile times and memory requirements, whereas their non-deterministic counterparts are expected to have a more compact representation, but as their name suggests, they are nondeterministic in their execution speed.

We compared the performance of our system against two software algorithms; one was a deterministic finite automaton (DFA) generator called `flex` and the other was `nrgrep`, which is essentially a simulated nondeterministic suffix automaton. (See Materials and Methods for the appropriate references.)

When regular expressions are input, `flex` compiles C code for the appropriate DFAs that are constructed to match the expressions. In turn, this code must be compiled into executables that can then be run to scan for the specified patterns in a data stream. `Nrgrep` uses a technique called bit-parallelism to simulate automaton behavior instead of actually generating them.

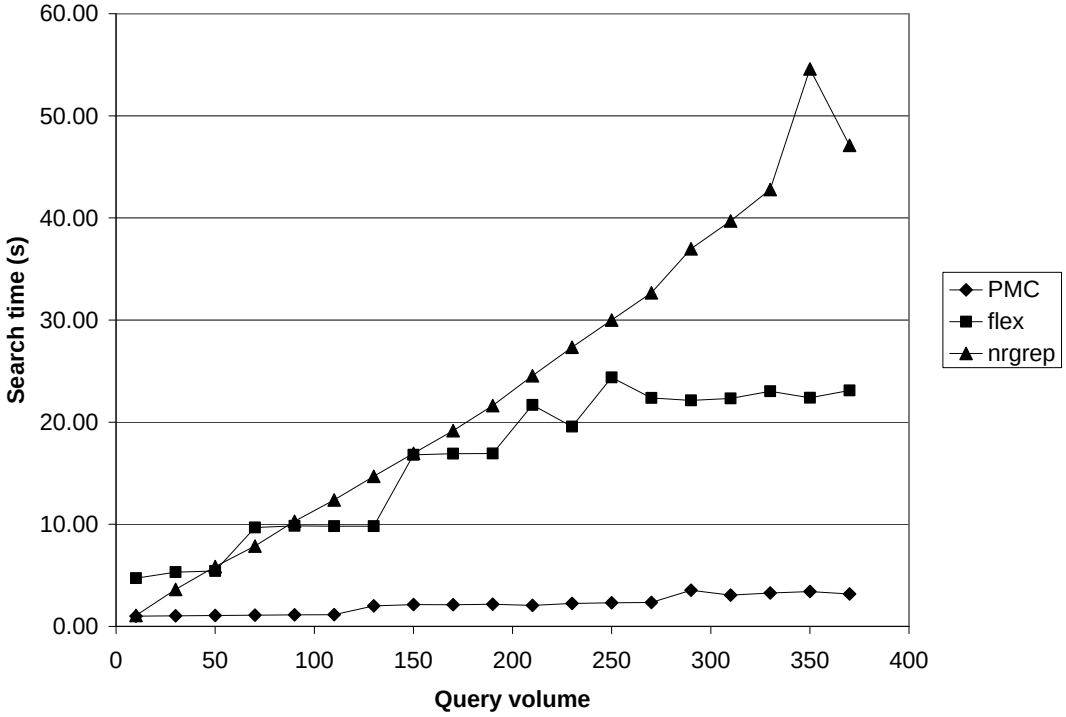


Figure 1. Search time versus query volume for different solutions.

4. Results

In our benchmarks, we risk comparing apples and oranges. For example, should we take into consideration the time it takes to compile the `flex` scanners or should we assume that the application is sufficiently static to render even `flex` compilation negligible eventually. Even though this may not necessarily be true in a real spam filtering solution, we have assumed that dynamic updating of queries will not affect the performance of the `flex` algorithm. Figure 1 shows the search time versus query volume for the PMC, `flex`, and `nrgrep`, respectively. The test was run using 10 duplicate entities of the dataset (see Materials and Methods), but the result is similar when using 1, 2, and 5 entities.

In our benchmark, we see that scalability is the main obstacle of `flex` and `nrgrep`. As the figure shows, we are able to keep a low gradient with respect to the query volume, in contrast to the other algorithms. Typical query volumes in a real-time application hosted by an internet service provider is about 10,000 queries, which emphasizes the need to be able to scale well in that dimension.

Figure 2 shows how the algorithms compare with the PMC solution when scanning with 250 queries in 1, 2, 5, and 10 times the data volume of the downloaded spam examples. As shown, the `nrgrep` and `flex` algorithms run into a much steeper gradient with respect to the dataset size than do the PMC, which is important considering that an internet service provider needs to scan hundreds if not thousands of emails per second.

To summarize, our PMC-based solution scales better than `flex` and `nrgrep` both with respect to query volumes and dataset size. Even though it appears that the `flex` algorithm is the closest competitor in the above benchmarks, it will probably be impractical in production systems due to

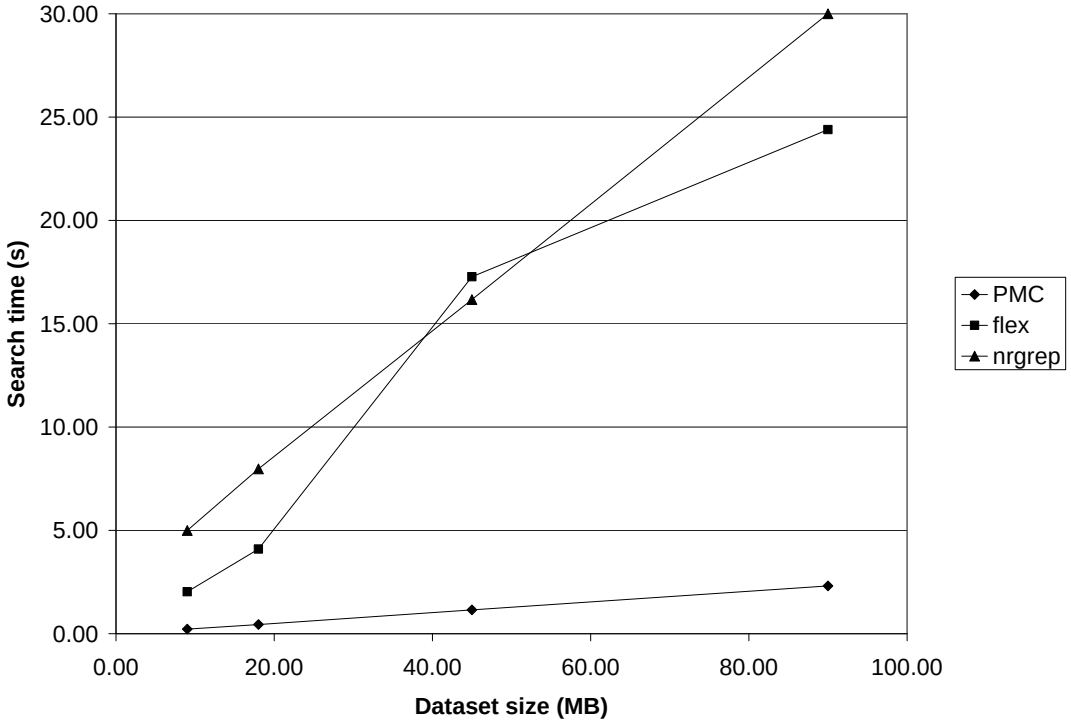


Figure 2. Search time versus dataset size for various solutions.

a lengthy and unpredictable compilation process. In the above benchmarks, we used only patterns that `flex` could handle in order to get a fair evaluation, but we experienced that `flex` broke down completely on several occasions. Consider the following listing, which is the first part of a query that `flex` were not able to handle:

$$(\text{earn}|\text{make}).\{1,20\}[0-9][0-9][0-9]+.\{1,30\}\ldots$$

Here, repetition of wildcards succeeds either of the words “earn” or “make”. Note that the “e” in “make” could mean the beginning of “earn” and if a number occurs it could either be a wildcard or one of the three or more numbers that are to follow the wildcards. Thus, the pattern cannot easily be represented by a DFA, as illustrated by the program’s performance breakdown. Patterns such as these were removed from the queries that comprised the benchmark set, even though neither the PMC nor the `nrgrep` algorithm experienced performance problems due to these patterns. A spam filtering must therefore either use other algorithms than `flex` or avoid rules that affect its performance at the potential expense of lower spam filtering performance.

5. Discussion

We have introduced a special-purpose search processor into a spam filtering solution to achieve higher performance and better scalability. As shown, the solution scales well in both the query volume and dataset size dimensions. In addition to higher screening performance and better scalability, our solution has the advantage of having negligible configuration processes compared with

the lengthy and unstable compilation of DFAs by `flex`. The `nrgrep` simulates a nondeterministic automaton, but does not generate it explicitly, which is why we are unable to compare its compilation performance to that of `flex`. Note, however, that the scaling performance is slightly worse than that of `flex`, which is not surprising considering that the `flex` numbers were obtained with precompiled scanners.

There is also substantial room for improvements in our solution. First, we can achieve six times the performance of these benchmarks by adding more PCI cards to the workstation. In these tests, we used a single card even though up to six cards can easily be fitted into a standard workstation provided there are enough PCI slots. Furthermore, the PMCs's lookup table (LUT) can be used to map any byte value from the input stream to another [3], which means that several patterns can be collapsed into a single expression. For example, `viagra`, `VIaGrA`, `v|agrA`, and so on can all be matched by `viagra` if uppercase letters are mapped to lowercase and `|` are mapped to `i`. Mapping schemes can be developed for characters that are often used to rewrite trigger words to avoid automatic filters that rely on correct spelling, but to remain easily comprehensible to the human eye.

6. Materials and Methods

6.1. Dataset

We downloaded data from the public spam repository SpamArchive.org. The set comprised 1,705 emails totaling 8,998,303 bytes of data, and a compressed version can be downloaded at `ftp://spamarchive.org/pub/archives/submit/691.r2.gz`. To test the methods's scalability with respect to data volumes, we concatenated two, five, and ten entities of this dataset.

6.2. Algorithms

As described in the main text, we benchmarked the performance of our PMC system against the `nrgrep` and `flex` algorithms. The fast lexical analyzer generator (`flex`) is a free implementation of the well-known `lex` program with some new features (see `http://www.gnu.org/software/flex/` for details, including a manual).

Nondeterministic reverse grep (`nrgrep`) is a member of the `grep` family of search algorithms, and uses bit-parallelism to simulate a nondeterministic suffix automaton. See the excellent book by Navarro and Raffinot for a detailed treatment of DFAs, NFAs, bit-parallelism, and the use of these concepts in regular expression matching [7].

6.3. Hardware

We used a standard workstation with an Intel Pentium 4 2.8 GHz processor with 1,024 MB DDR-SDRAM running the Woody release of Debian Linux (see `http://www.debian.org`). Tests that involve special purpose search processors were run on the same workstation with a single PMC card installed.

References

[1] Doug Burger, James R. Goodman, and Alain Kägi. Memory bandwidth limitations of future microprocessors. In *23rd International Symposium on Computer Architecture (ISCA)*, pages 78–89. IEEE Computer Society, May 1996.

[2] Wu-chun Feng. Green destiny + mpiblast = bioinfomagic. In *ParCo 2003*, pages 653–660, 2003.

[3] Arne Halaas, Børge Svingen, Magnar Nedland, Pål Sætrom, Ola Snøve Jr., and Olaf René Birkeland. A recursive MISD architecture for pattern matching. *IEEE Trans. on VLSI Syst.*, 12(7):727–734, 2004.

[4] W. Daniel Hillis and Lewis W. Tucker. The CM-5 connection machine: a scalable supercomputer. *Communications of the ACM*, 36:31–40, 1993.

[5] Graham Kirsch. Active memory: Micron's Yukon. In *17th International Parallel and Distributed Processing Symposium (IPDPS)*, page 89. IEEE Computer Society, 2003.

[6] N. R. Adiga et al. An overview of the BlueGene/L supercomputer. In IEEE, editor, *SC2002: From Terabytes to Insight. Proceedings of the IEEE ACM SC 2002 Conference*, 2002.

[7] Gonzalo Navarro and Mathieu Raffinot. *Flexible pattern matching in strings: practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, Cambridge, UK, 2002.

[8] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A case for intelligent RAM: IRAM. *IEEE Micro*, 17(2):34–44, Apr 1997.

Implementing Critical Sections with the Shared Explicit Cache System in the Shared Memory Parallel Architectures

Tomasz Madajczak[a b], Henryk Krawczyk[a]

[a]Gdansk University of Technology, Faculty of Electronic, Telecommunication and Informatics, Dep. of Computer System Architectures, ul. Narutowicza 11/12, 80-233 Gdansk, Poland

[b]Intel Technology Poland Ltd. ul. Slowackiego 173, 80-247 Gdansk, Poland

1. Introduction

This document presents the new method for implementing critical sections in the shared memory parallel architectures such as multithreaded multiprocessors integrated on a die. The method bases on the Shared Explicit Cache System (SHECS) implemented in the multiprocessor. The method is derived from the Folding method [1][2] and is also similar to the cache-based synchronization technique proposed in [3][4]. The new method in comparison with the state-of-the-art methods eliminates their limitations and disadvantages such as scalability, starving, while it adds usage flexibility and copes with the hardware multithreading extensions. The document presents the system architecture equipped with the SHECS and also the algorithm to implement operating system or application level locking service.

2. Background

2.1. Folding method in network processors

Efficient use of shared resources is always connected with the need of a critical section implementation. Multicore and multiprocessors systems have the ability to rely on specific hardware support and capabilities to synchronize the particular processor cores within the die or the integrated platform. Hardware techniques for critical sections are available in the network processors and they are very efficient, but not always universal [1]. Software techniques are still available for the SMP systems and for cases when the hardware support isn't flexible. Therefore, there is a need for an efficient and flexible method that would address the new perspectives for the shared memory parallel architectures such as multithreaded multiprocessors systems. This document presents such a new method based on the use of SHECS. It is derived from the Folding method.

The Folding method was introduced in the Intel's network processors IXP 2000 [5] as a universal method for programming software critical section for the threads of a single microengine (that is a RISC processor). The method uses the microengine's internal local memory and CAM (content addressable memory) lookup engine that comprises an internal explicit cache system managed with software.

Folding caches the read data to be modified. It manages with the following critical section scenario that firstly reads a resource, then modifies it, and finally writes back the modified data. The read resource is stored within this system and considered locked if its use counter is greater than 0. Folding may occasionally lost the order of the threads entering the critical section, because in case of finding locked entry the algorithm repeats the section entering. This order lost means that the critical section may be starving for some threads, as they have no luck and they always repeat entering. Extending the Folding method onto a number of processors is not an easy task and also starving critical sections aren't useful in the general purpose parallel systems.

All these issues are solved in the SHECS-based method. It bases on the new explicit cache memory system architecture that eliminates the Folding's limitations and disadvantages. Thus, the SHECS-based method is more universal, flexible, and may have more applications.

2.2. Cache coherency

The parallel shared memory architectures built with using general purpose processors with internal caches may have implemented a mechanism for enforcing the coherence of internal caches. Hardware solutions for this problem are presented in [6][7], while software algorithms are discussed in [8][9]. Generally there are two approaches: implicit methods that hide the problem for the system user or software, and explicit methods assuming that the system user or software is aware of the problem, treats cache as a normal shared resource and solves it with cache-locking [4] or other locking method. The introduced method addresses the problem in the similar way as explicit methods. It assumes explicit cache-locking with integrated data transfers of the most recent cached data value and additionally it copes with hardware threading.

3. Shared Explicit Cache System

3.1. Architecture assumptions

Shared explicit cache system (SHECS)[1] consists of CAM banks controller, a number of CAM banks and some shared fast SRAM memory for caching. Figure 1 shows that the shared explicit cache system should be connected in a similar way as shared memory to the parallel processors P1-PN. The key assumption is providing a number of CAM banks that can work together or separately.

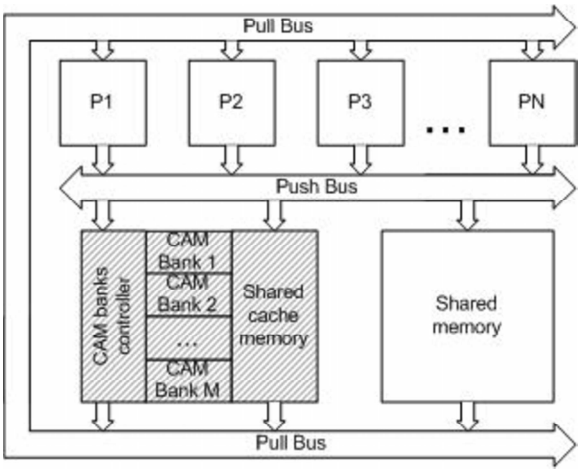


Figure 1. The SHECS in a shared memory parallel architecture

Every CAM bank is a functionally complete unit comprising entries tag and lock latches, lookup-locks FIFO queue and configuration for associated region of the cache memory. Every operation on the explicit cache system should have associated the mask that determines which banks are associated to the operation (single bit in the mask controls (enables/disables) one CAM bank). From the high level point of view the masking capability allows coupling a number of independent CAM

[1]The concept of the shared explicit cache system for network processors is patent pending in the U.S. patent office.

banks into a single CAM. Such a single CAM should also have a consistent old tag removing policy - least recently used tag should be removed upon adding a new tag within all the banks enabled with a particular mask. This requirement is realized with the CAM banks controller logic. The method of enforcing coherency of critical sections is hardware signaling. The signal arrival wakes up a virtual or hardware thread that normally waits on that signal for the completion of CAM-lookup-lock operation.

3.2. Functionality

A functionally complete SHECS should provide at least the following operations within the specified banks:

- CAM lookup with data locking and integrated reading
- CAM entry unlocking
- CAM entry's cache reading and writing with entry unlocking
- CAM banks managing (clearing the bank, adding, deleting tag, etc.)

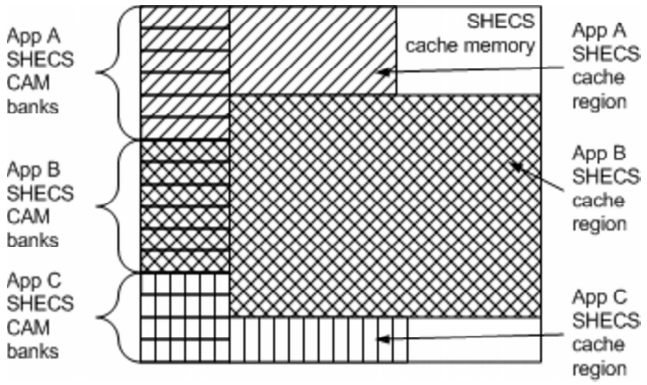


Figure 2. Example of the SHECS allocation among three parallel applications

The key feature of SHECS is ability to flexible divide onto independent subunits. This feature allows implementing a system wide SHECS allocation manager that on request allocates the specified amount of CAM entries. This capability is illustrated in Figure 2.

Every CAM bank may have associated a line of cache memory to store data to be locked and modified. Because, the memory is a part of the explicit cache memory system, the reading of such a memory line may be integrated with a successful CAM lookup operation. If processors P1-PN have a support for reading data bursts directly into the registers (such as network processors) this data may be used at once. Another way to use CAM lookup with integrated data transportation is to store data bursts within the processors' local caches. In that case the local cache may use the tag with the local processor address of this particular shared cache memory region or calculate the tag from the CAM entry index. The SHECS may handle some number of pending simultaneous CAM-lookup-locks. This limitation is connected with the depth L of the FIFO queue storing pending lookup-locks. Such a FIFO is implemented in the each CAM bank. The depth L may be exceeded

if a parallel application performs more than L simultaneous CAM-lookup-locks for the same tag value. In that case the CAM-lookup-lock requests are rejected with the result indicating locking fullness. In the other words the locking is not starving as long as the number of requesting task is lower and equal to the depth L. This depth should be chosen for a parallel system with considering the following factors:

- The number of N of parallel processors
- The number of K of hardware threads in each processor

The value of L should be calculated with the following formula:

$$L = N * K * C \tag{1}$$

where: N - the number of processors; K - the number of hardware threads in each processor; C - some constant $>= 1$

The value of depth L implemented according to formula (1) manages with all synchronization problems that use maximally all available parallel resources. However if a program tries to be executed with higher level of parallelism than available parallel resources (in the other words the program have more virtual threads or processes than there is hardware threads in the parallel system), then the locking implemented with using the explicit cache system may be starving for some virtual threads.

4. Implementing SHECS-based Full-Locking Critical Sections

Full-locking technique (described well in [10]) assumes that every element that must be used or modified coherently may be locked independently for a certain time. Because the explicit cache system has hardware limitations such as limited number of CAM entries and limited number of pending locks for a particular CAM bank (due to the limited depth L), the implementation of full-locking critical sections should combine the explicit cache system functionality and some OS (operating system) mechanismism to queue rejected SHECS requests. Otherwise, if only using the hardware technique, the parallel program decomposition should allow starving, that may happen if the program has more threads or processes than the depth L that want to coherently use the same resource.

A non-starving implementation with optional OS support is depicted in Figure 3 and works as decribed below. The critical data stored within the SHECS is considered locked if the lock bit is set for the relevant CAM entry tag. Otherwise if the lock bit is clear, it is considered unlocked and may be reused (it is coherent). If the critical data is not stored within the SHECS it is obviously unlocked. A thread attempting a critical section performs a lookup-lock operation in the explicit CAM system using a data identifier (index or address).

1. CAM miss means that the critical data is unlocked. The SHECS reserves (locks) a least recently used entry and writes its tag with the data identifier, then returns the entry's index within the result miss-entry-reserved, if such an entry is available. If not (all entries are locked within the specified banks and there isn't any entry to be reserved)- the SHECS returns status miss-all-reserved.

    - If the result is miss-entry-reserved the thread must read the critical data from the shared main memory. It should update the CAM entry's cache with the modified data upon exiting from the critical section with CAM-cache-write-unlock operation. After the update,

the critical data resides in the cache. If there were any CAM-lookup-lock operations for the same critical resource in the meantime, they were queued in the lookup-lock FIFO queue in the same CAM bank. In that case the first candidate for entering the critical section is released from the FIFO and signaled to enter the section.

- Otherwise (miss-all-reserved), the thread should be queued by the OS in a software lock-pending queue that also means that it has been swapped out in a software way.

2. CAM hit with detecting lock bit set means that the critical data is locked. In that case the operations are queued in the lookup-lock FIFO queue in the same CAM bank if the FIFO has the available space. In that case the thread is swapped out as long as it will be removed from the FIFO and signaled upon the leaving of critical section by another thread. Otherwise, the result is hit-locked and in that case the OS queues the thread in a software lock-pending queue that also means that it has been swapped out in a software way.

3. CAM hit with detecting lock bit clear state means that the critical data resides in the hit entry's cache and it is unlocked. In that case the lock bit for this CAM entry is automatically set and the critical data with the result hit-unlocked is transported from the cache to the thread registers (feature supported in the network processors) or to the internal cache of the processor executing the thread. After using and modifying, the critical data should be updated in the SHECS's cache and in the main shared memory. The cache is updated with operation CAM-cache-write-unlock that automatically unlocks the CAM entry and consequently if there were any CAM-lookup-lock operations for the same critical resource in the meantime, they were queued in the lookup-lock FIFO queue in the same CAM bank. In that case the first candidate for entering the critical section is released from the FIFO and signaled to enter the section.

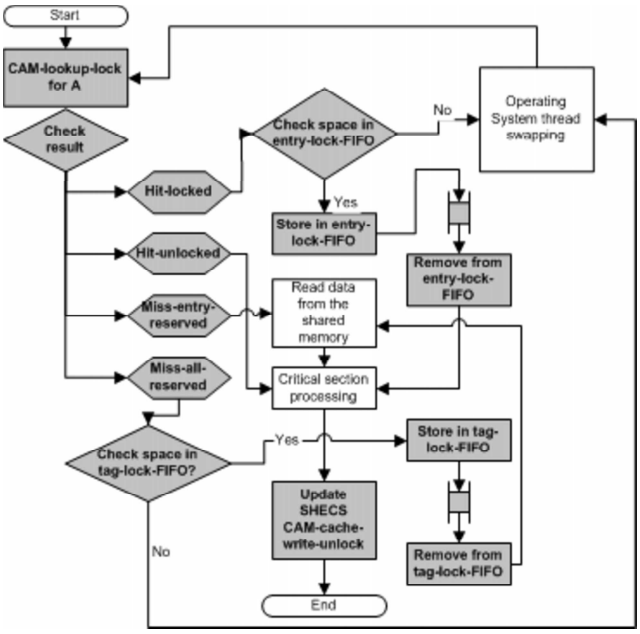


Figure 3. The SHECS's algorithm for full-locking critical sections (gray elements mark hardware processing)

5. SHECS's Simulation on Network Processor

5.1. Simulation method

The SHECS was simulated with the Developer Workbench 4.2 for Intel's IXA Network Processors. Two microengines were used to simulate two SHECS banks with capacity of 32 cacheable entries in total. The latency introduced with the SHECS simulation was assumed comparable with hardware implementation latency. The parallel application was running on four microengines and it implemented the full-locking locking critical section. Every thread in the application performed random data accesses to a table with limited number of entries. The application's algorithm is illustrated with the below pseudo code:

Parallel application pseucode

```
while (no_loops--) {
   random= PSEUDO_RANDOM_NUM mod 32; //or mod 128
   calculate addrA from random;
   cycle_delay(cycle_delay1);
   shecs_lookup_lock(/*in*/addrA, /*out*/entryA, /*out*/indexA);
      modify entryA;
      cycle_delay(cycle_delay2);
   shecs_write_unlock(/*in*/addrA, /*in*/entryA, /*in*/indexA);
}
```

The critical section is between `shecs_lookup_lock()` and `shecs_write_unlock()` primitives. Time spend in the critical section was chosen to be 0.1 of the processing time before it. Thus the critical section was relatively short. The experiments were made with varying the following parameters:

- number of entries in the shared table - 2 values: 32 - equal to the SHECS's capacity, 128 - four times larger than the SHECS's capacity
- number of executing threads on four microengines, values were 32, 16, 8, 4 that were 8, 4, 2, 1 threads per one microengine respectively

5.2. Simulation results

The SHECS possible results were:

- bypassed - critical data had been stored unlocked in the SHECS and after locking it was transfered to a new critical section owner
- locked-bypassed - critical data had been stored locked in the SHECS and after lock release it was transfered to a new critical section owner
- reloaded - critical data had not been stored in the SHECS and the entry's ownership was granterd to a new critical section owner
- locked-reloaded - all SHECS's entries had been used and after freeing one of them its ownership was granted to a new critical section owner

Figure 4 illustrates that the adding of parallel resources might reduce the scalability of speedup, because the number of pending locks increases. Figure 5 and Figure 6 show the distributions of results in cases when the SHECS's capacity is equal and is less to the number of locked shared resources, respectively.

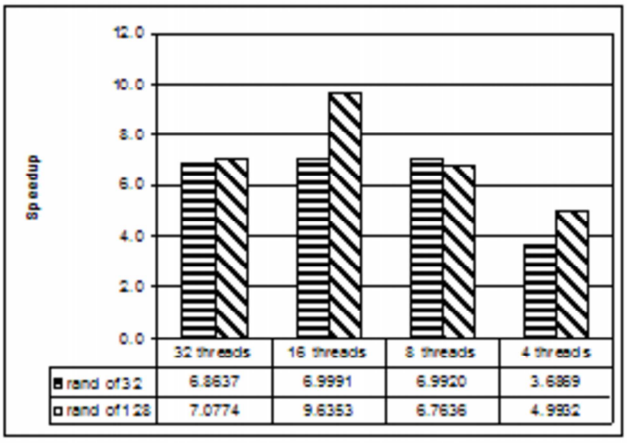


Figure 4. Speedup vs. number of parallel threads and different table sizes

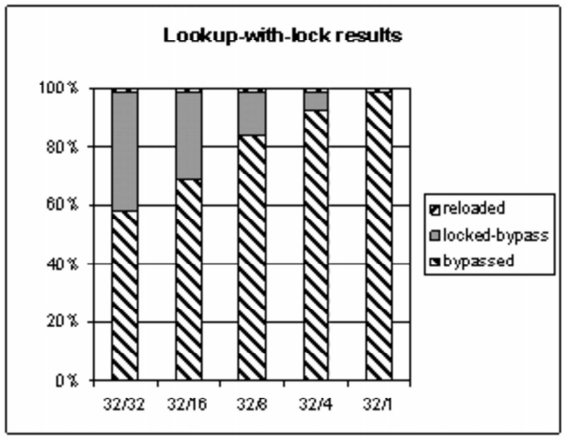


Figure 5. The distribution of results for execution with table size equal to the SHECS's capacity

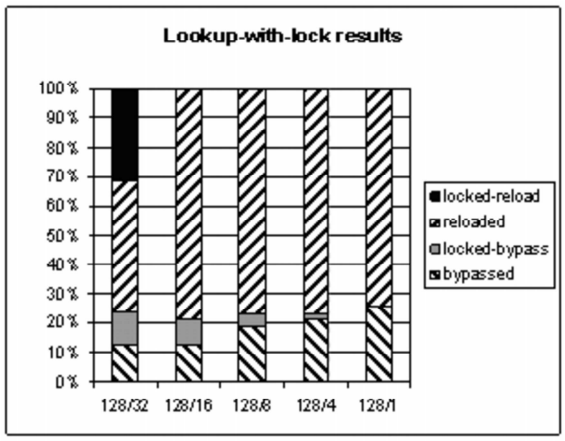


Figure 6. The distribution of results for execution with table size 4 times larger than the SHECS's capacity

6. Final Remarks

The use of SHECS in a parallel system with shared memory architecture should enable achieving the best possible performance gain in the data driven parallelization of sequential programs with required data coherency. The SHECS may be also used to speed-up data searching algorithms, thanks to providing explicit associative searching in CAM banks for a value. Therefore, the parallel algorithms, that search and modify shared dynamic data structures, can benefit from the both SHECS features critical section synchronization with caching data and associative searching. Such features combination constitutes very powerful proposition for the shared memory architectures. The SHECS increases the real parallelism in such systems and also proposes the critical sections support for managing the cache coherence problems. The only disadvantage of the SHECS is the cost - it is an additional, manageable cache system. The content addressable memories (CAMs) (the key ingredient of the SHECS) are still pricy and they aren't used in the general purpose systems. They are still perceived as rather co-processing elements in the network systems in which they are responsible for associative searching of a route for a packet. However the Moore's Law constantly decreases the cost of silicon circuits and it may cause that in the feasible future the SHECS will be implemented and will offer performance gain in the parallel multicore systems integrated on a die.

References

[1] H. Krawczyk, T. Madajczak: Optimal Programming of Critical Sections in Modern Network Processors under Performance Requirements. In the Proc. of IEEE Parelec Conf. Dresden 2004. 2004

[2] M. Adiletta, M. Rosenbluth, D. Bernstein, G. Wolrich: The Next Generation of Intel IXP Network Processors. Intel Technology Journal Vol. 6 issue 3. 2002

[3] N. Aboulenein, J. Goodman, S. Gjessing, P. Woest: Hardware support for synchronization in the Scalable Coherent Interface (SCI). In Eighth International Parallel Processing Symposium. 1994

[4] U. Ramachandran, J. Lee: Cache-based synchronization in shared memory multiprocessors. In Supercomputing '95. 1995

[5] M. Adiletta, D. Hooper, M. Wilde, Packet over SONET: Achieving 10 Gbps Packet Processing with an IXP2800. Intel Technology Journal Vol. 6 issue 3. 2002

[6] A. Nanda, A. Nguyen, M. Michael, and D. Joseph: High-Throughput Coherence Controllers. In the Proc. of the 6th Int'l HPC Architecture. 2000

[7] M. Azimi, F. Briggs, M. Cekleov, M. Khare, A. Kumar, and L. P. Looi: Scalability Port: A Coherent Interface for Shared Memory Multiprocessors. In the Proc. of the 10th Hot Interconnects Symposium. 2002

[8] A. Grbic: Assessment of Cache Coherence Protocols in Shared-memory Multiprocessors. A PhD thesis from Grad. Dep. of Electrical and Computer Engineering University of Toronto. 2003

[9] G. Byrd: Communication mechanisms in shared memory multiprocessors. A PhD thesis from Dep. Of Electrical Engineering of Stanford University. 1998

[10] R. Jin, G. Yang, G. Agrawal: Shared Memory Parallelization of Data Mining Algorithms: Techniques, Programming, Interface, and Performance. IEEE Transactions on Knowledge and Data Engineering, Vol. 16, No.10. 2004

Scheduling

A new Genetic Convex Clustering algorithm for parallel time minimization with large communication delays

J. Pecero Sanchez[a], D. Trystram[a]

[a]Laboratoire Informatique et Distribution (ID) IMAG, ZIRST 51 Av. Jean Kuntzmann, 38330 Monbonnot Saint Martin, France

1. Introduction

The efficient execution of an application on a parallel and distributed system highly depends on the decisions taken for scheduling the tasks that constitute the program. One solution for obtaining efficient parallel programs execution is task clustering. It corresponds to assign the tasks to clusters where each cluster is run on a single processor. For most of the new distributed platforms available today, communication delays are the main factor that affects the performance of these systems. Communications are usually relatively high in regard to basic execution time. The problem of clustering the tasks to be executed on a multiprocessor computer is one of the most challenging problems in parallel computing. There is no fully satisfactory results for the task clustering problem if large communication delay is considered.

We are interested here in the task clustering problem on an unbounded number of processors taking into account large communication delays. To solve it, we introduce a new genetic algorithm(GA) approach which has nice properties called Genetic Convex Clustering Algorithm (GCCA). The main idea is to assign tasks to locations in *convex* groups. We consider arbitrary execution time. We propose a novel crossover operator in the context of the task clustering problem.

The remainder of this paper is organized as follow. Section 2 presents the task clustering problem. An analisys of convex clusters properties will be presented briefly in Section 3. Section 4 introduce the GCCA, which considers the convex cluster properties. Section 5 evaluates the performance of the GCCA . Finally, Section 6 concludes this paper.

2. Task Clustering

Let us start by some preliminaries. As it is common, an application to be parallelized is represented as a precedence task graph. A precedence task graph is a weigthed acyclic digraph (DAG) *G=(V,E)*, where *V* is the set of tasks nodes, which are in one-to-one correspondance with the computational tasks in the application and *E* represents the set of the precedence relations between the tasks.

Clustering is often used as a compile-time preprocessing step in mapping parallel programs onto multipocessor architectures. In this context, given a precedence tasks graph and infinite number of fully connected processors, the objective of clustering is to assign the tasks to processors. The task clustering problem is *NP-Hard* [1][2], even when the number of processors is unbounded and tasks duplication is allowed. For most of the new parallel and distributed platforms available today, communication delays are the main factor that affects the performance. There is no fully satisfactory results for the task clustering problem if large communication delay is considered despite the result of Papadimitriou and Yannakakis [3], who proposed a constant approximation ratio when replication of tasks is allowed. If no replications are allowed, the problem of clustering with large communication delays has no satisfactory solution today. Thus, practical solutions have been proposed, based mainly

on three different heuristics: the algorithms based on the critical path analysis [1][4], priority-based list scheduling [5] and based on graph decomposition [6][7].

Another heuristic method used in the task clustering problem and scheduling context is the meta-heuristic known as Genetic Algorithm(GA) [8][9][10][14]. A genetic algorithm is a guided random search method where elements (called *individuals*) in a given set of solutions (called *population*) are randomly combined and modified (these combinations are called *crossover* and *mutation*) until some termination condition is achieved.

In this paper we introduce a new genetic algorithm approach which has nice properties called Genetic Convex Clustering Algorithm (GCCA). The main idea and the force of GCCA approach is to assign tasks to locations in convex groups. In the following sections we will detail this approach. First, we will analyze the convex cluster properties, after that the GCCA will be presented.

3. Analysis of Convex Cluster

In this section, we describe the convex cluster algorithm. The main idea is to assign tasks to locations in convex groups. We consider arbitrary execution time and large communication delays.

In the following, as introduced in Section 2, we consider a precedence tasks graph for representing the application to be parallelized. Let *G=(V, E)* denote such a graph. Let $\prec$ be the partial order of the tasks in G, its inverse relation $\not\prec$ and $\sim$ the equivalence relation defined as follows: for x and y $\in$ *V*, $x \sim y$ if, and only if $x \not\prec y$ and $y \not\prec x$. If two tasks x and y are such that $x \sim y$ we say that x and y are independent.

Definition 1 *A group of tasks* $T \subset V$ *is said* $convex$ *if and only if all pairs of tasks* $(x, z) \in T$*, all intermediate task y such that* $\forall$ *y,* $x \prec y \bigwedge y \prec z \Longrightarrow y \in T$ [7].

An example of convex cluster is depicted on Figure 1. A precedence task graph is depicted on left of Figure 1. Each node label shows the number of task and each edge label shows intertask communication cost between tasks. Each label beside of the nodes represents the task computation cost. The intra-communication cost (inside each cluster) on the clustered graph is zero.

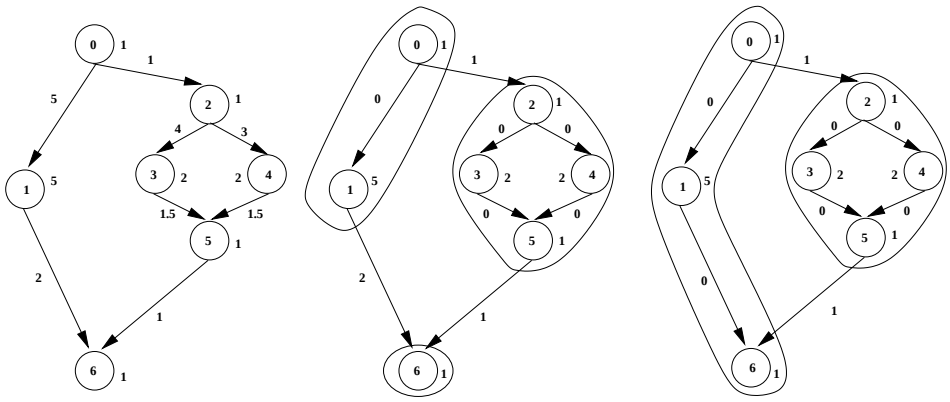


Figure 1. On left, a precedence task graph is depicted. In the middle, the graph is clustered on a valid convex cluster. On the right, we have a clustered graph but it is not convex.

The convex clustering has the following properties:

Property 1. The convex cluster is a particular class of $Processor\ Ordered\ Schedule$ [11], it means, if a cluster is assigned to processor i, the predecessors of all tasks belonging to this cluster

are into the processors with index less than i.

Proof. The proof is by contradiction. Let assume that a cluster C_i is not convex and is assigned to processor with index i, then there exist *x* and *z* $\in C_i$ and $y \in C_j$ assigned to processor with index $i+1$ such that $x \prec y \prec z$. As $x \prec y$, there exists a path between processors i and $i+1$, and similarly there exists a path between processors $i+1$ and i because $y \prec z$. It is impossible by definition of convex cluster and by definition of processor ordered schedule.

Proposition 2. The resulting clustered graph in a convex cluster is a Directed Acyclic Graph.

Proposition 3. This class of algorithm with the convex cluster properties and taking into account unit execution time and large communication delays was proved to be *2-Dominant*, that is, there exists a convex clustering whose execution time on an unbounded number of processors is less than twice the time of an optimal clustering.

So, this makes a good solution for solving practical scheduling algorithms since we expect to be able to schedule more easily convex clusters than general graphs.

Now that we have shown the strength of convex clustering, it remains to find a "good" convex clustering. For this problem in [7] proposed an algorithm based on a recursive decomposition of the precedence task graph. This algorithm does not allow to obtain all the convex clusters. The problem of convex cluster based on a recursive decomposition is NP-Complet [13]. So, to build all the convex cluster we propose to use genetic algorithm.

4. Genetic Convex Clustering Algorithm

Genetic Algorithm (GA) is a guided random search algorithm based on the principles of evolution and natural genetics. It combines the exploitation of past results with the exploration of new areas of search space. By using survival of the fittest techniques and a structured yet randomized information exchange, genetic algorithm can mimic some of the innovative flair of human search. Genetic algorithm is randomized but not simple random walks. It exploits historical information efficiently to speculate on new search points with expected improvement [12].

The research on using GAs for clustering and scheduling tasks in parallel computing is an active research, with many papers showing the potential use of this class of algorithms [10][14]. Each approach contains its own characteristics, but the main difference of these algorithms is the representation of a solution. In this section, we describe our proposed approach.

4.1. Algorithm design

The Grouping Genetic Algorithm is a genetic algorithm heavily modified to suit the structure of grouping problems. Those are the problems where the aim is to find a good partition of a set, or to group together the members of the set [15]. GCCA belong to this class of algorithm. To represent a valid solution and to handle the genetical operators (i.e crossover) with convex cluster, we proposed the following representation.

Coding

A solution representation of a task clustering problem based on convex cluster is feasible if it satisfies the following conditions:

1. Each task of the precedence tasks graph belongs to exactly one cluster.
2. The restriction of convexity must be fulfilled.
3. The fitness function of GCCA depends only on the clustering of the tasks, rather than the numbering of the cluster.

To represent a valid convex cluster, we propose a bichromosomal representation. The first chro-

mosome encoded the number of tasks and the second chromosome encoded the number of groups. We augmented this representation with a group part, that is, the group part encoding the cluster on a one task for one cluster basis. The length of the bichromosomal representation is equal to the number of tasks and the length of the group part representation is variable and this one depends on the numbers of clusters. The Figure 2 depicts a valid solution representation, meaning the tasks 0 and 1 are assigned in cluster 0, the tasks 2, 3, 4 and 5 in cluster 2 and finally the task 6 is assigned to processor 3. The group part of the chromosome represents only the groups(clusters), in this case, express the fact that there are three clusters in the solution. The main fact is that the genetic operators will work with the group part of the chromosomes, the standard task part of the chromosomes identify which items actually form which group.

Figure 2. A chromosome representation of the convex clustered graph depicted in the middle of the figure 1.

Initial population

The first step in GCCA is to create an initial population. Most of the genetic algorithms applied to the task clustering problem create a random initial population. We propose an algorithm to generate the initial population. This algorithm was adapted to build feasible convex clusters. The algorithm is the following:

1. Put all the tasks in a list called ListTasks.
2. Choose randomly a task of ListTask and assign it a cluster.
3. Compute the predecessors set and the successors set of the selected task.
4. Eliminate these tasks of the ListTasks.
5. **While** (ListTasks != $\emptyset$)

5.1 Choose randomly an independent task and to assign it a different cluster.

a). Compute the predecessors set and the successors set of the selected task.

b). Eliminate these tasks of the ListTasks.

EndWhile

6. Copy the predecessor sets in a global predecessor set.
7. Assign into a cluster all the tasks that are duplicated in the global predecessor set.
8. Eliminate all the duplicated tasks of the corresponding set.
9. Copy the successors sets in a global successor set.
10. Eliminate all the duplicated tasks of the corresponding set.
11. Assign the remaining tasks of the predecessor and successor sets in the correspondant cluster.

Evaluation

The genetic algorithm aim is to maximise the fitness function, while minimising the objective function. To evaluate the fitness of each solution, first the schedule length of the particular solution is determined and after that, this schedule length is mapped to an initial fitness value (f_i') using equation (1) [9]:

$$f_i' = sumofallthetasksexecutiontimes - schedulelength. \quad (1)$$

Where the *sum of all the tasks execution times* is defined as the time it would take to run all the tasks in the DAG on a single processor. However, in some cases f_i' can still be negative since some of the initial clusterings can result in schedules whose length is worse than running the tasks on a single processor. So a linear scaling is employed to obtain the final fitness value (equation (2)).

$$f_i = \frac{f_i' - MIN[f_i']}{MAX[f_i'] - MIN[f_i']} \tag{2}$$

To obtain the schedule length for a particular clustering each task is scheduled in decreasing order priority. That is, schedule a task into the first slot available after the specified earliest start time on the assigned processor. The earliest start time $e(v)$ for task v is calculated using the equation (3):

$$e(v) = MAX[s(u) + \mu(u) + \lambda(u,v), (u,v) \in E] \tag{3}$$

Where: $s(u)$ is the starting execution time of task u, $\mu(u)$ represents the execution time of task u, $\lambda(u,v)$ represents the tasks communication time. $\lambda(u,v) = 0$ if tasks u and v are placed on the same processor. Once all tasks have been scheduled, the algorithm uses the equation (4) to determine the schedule length of the scheduled DAG.

$$MAX[s(v) + \mu(v)] \tag{4}$$

Stop condition

The obvious way to stop the genetic algorithm would be to wait until the optimum is found, but due to the complex solution spaces there is no indication whether the optimal or only a near-optimal solution has been located. So, there are three ways GCCA will terminate. Firstly, if it has reached the maximum number of iterations. Secondly, when the best fitness in a population has not changed for a specified number of generations. Finally, when the difference between the average and the best fitness remains constant for some given number of generations.

4.2. Genetic operators

Selection

Selection uses a proportionate selection scheme and replaces the entire previous generation. The proportionate selection scheme where a string with fitness value f_i is allocated to a relative fitness of f_i/f_{aver}, where f_{aver} is the average fitness of the population. GCCA uses the *roulette wheel* style of selection to implement proportional selection. The algorithm selects strings until the next generation is completely generated.

Crossover

Crossover produces new individuals that have some portions of both parent's genetic material. As pointed out in the previous section, GCCA crossover will work with variable length chromosomes with genes(group part) representing the clusters. To generate a valid solution we used the following algorithm:

1. Select randomly two crossing sites, delimiting the *crossing section*, in each of the two parents.
2. Inject the contents of the crossing section of the first parent in the first crossing site of the second parent. Recall that a crossover works with the group part of the chromosome, so this means injecting some of the *clusters* from the first parent into the second.
3. Eliminate all *tasks* now occurring twice from the clusters they were members of in the second parent. Consequently, some of the old groups coming from the second parent are altered: they do not contain all the same tasks anymore, since some of those tasks had to be eliminated.

4. If necessary, *adapt* the resulting clusters, according to the convex cluster constraints.

5. Apply the points 2. through 4. to the two parents with their roles permuted in order to generate the second child.

Considering the graph depicted in Figure 1, a crossover example is depicted in Figure 3.

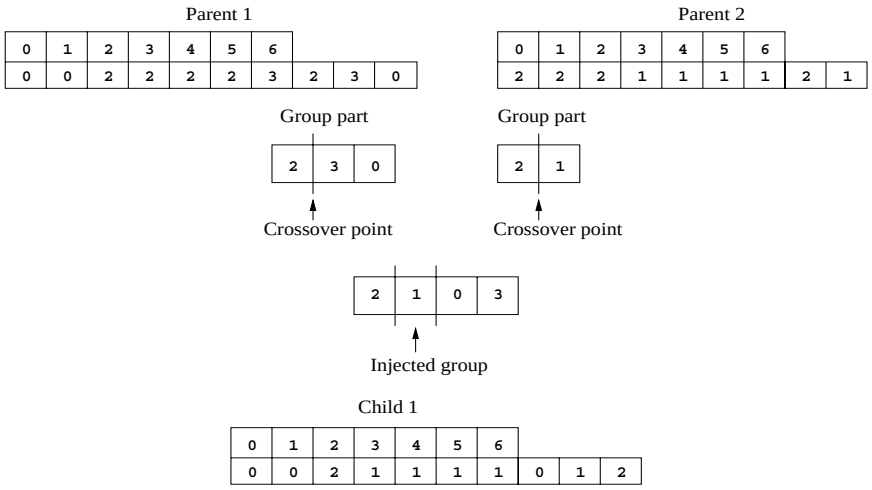


Figure 3. A crossover example.

4.3. Mutation

The main objective of mutation operation is to produce a slight perturbation in the search, in order to quit a local minimum. The GCCA mutation works by changing a task's cluster (*if it is possible*) to any cluster other than the one it was originally.

5. Results

In producing the results given in this section, the following benchmark values were adopted: population size of 100, crossover probability of 0.8, mutation probability of 0.01, a generation limit of 100, unlimited number of processors.

To evaluate the performance results of GCCA, we used a benchmark of *Reference Graphs* (RG). The Reference Graphs are tasks graphs that have been previously used by different researchers and addressed in the literature. This set consists in about 9 graphs (7 to 18 task nodes). These graphs are relatively small graphs but do not have trivial solutions and expose the complexity of clustering very adequately.

Figure 4 shows the performance of GCCA in a graph of 10 tasks. The task graph used here has an optimal schedule length of 26 unit times and 2 clusters [9]. On left of Figure 4, the convex clustering obtained by GCCA is showed. In the middle, the Gantt chart (a graphical representation of the duration of tasks against the progression of time) represents the best solution performed by GCCA. On the right, the Gantt chart depicts the optimal schedule reported in the literature. GCCA obtains the optimal schedule length but the number of processors is greater than the number reported in [9] which has only 2. It is due to the fact that GCCA tries to use all the available processors.

Table 1 summarizes the experiment results obtained by GCCA. The first column shows the reference of the graphs. The number of tasks in the RG are shown in the second column. Third and fourth

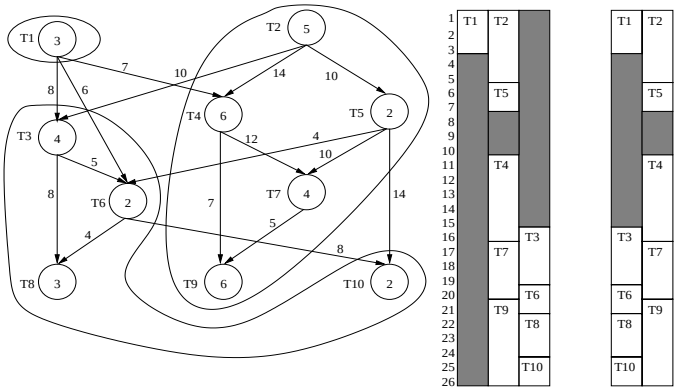


Figure 4. Convex cluster solution to the problem of 10 tasks. The tasks are represented in white, idle times are represented in gray.

References	No. nodes	Optimal schedule	Optimal clusters	GCCA schedule	GCCA clusters
[9]	7	9	2	10	3
[18]	7	4	2	4	4
[9]	9	5	2	5	5
[20]	9	149	2	160	3
[9]	10	26	2	26	3
[19]	10	11	2	10	6
[17]	13	190	2	190	5
[16]	18	440	3	480	5
[16]	18	330	3	340	6

Table 1
Table of results.

columns represent the optimal schedule length and the optimal number of clusters reported in the literature, respectively. Finally, the best schedule and the best number of cluster obtained by GCCA are reported in the two last columns of the figure. The experiment results show that GCCA is able to find optimal solutions for some test cases. In some other cases the number of clusters computed by GCCA is greater than the optimal number of clusters reported in the literature.

6. Conclusions

In this work we presented a new genetic algorithm called Genetic Convex Cluster Algorithm, which has nice properties. This algorithm was applied to solve the task clustering problem with large communication delays. The experiment results obtained by GCCA show the feasibility of using genetic algorithm with the convex cluster properties to solve the task clustering problem. The GCCA approach seems particularly well suited for the new parallel systems like cluster of PC with hierarchical communications. We proposed two novel genetic operators (representation and crossover) in the context of the task clustering problem.

References

[1] V. SARKAR (1989). *Partitioning and Scheduling Parallel Programs for multiprocessors.* Pithman, 1989.

[2] P. CHRÉTIENNE AND E.G.COFFMAN AND J.K.LENSTRA AND Z.LIU (1985). *Scheduling Theory and its Applications*, chapter *Scheduling with Communications Delays: A Survey.* John Wiley and Sons (1985), New York.

[3] C. H. PAPADIMITRIOU AND M. YANNAKAKIS (1990). *Towards and architecture independent analysis of parallel algorithms*. SIAM Journal on Computing, vol. 19. pg. 322-328.

[4] A. GERASOULIS AND T. YANG (1993). *DSC: Scheduling parallel tasks on an unbounded number of processors*. IEEE Transaction on Parallel and Distributed Systems, vol. 4, num. 6, pag. 686-701.

[5] J.J. HWANG AND Y. CH. CHOW AND F. D. ANGERS AND CH. Y. LEE (1989). *Scheduling precedence graphs in systems with Interprocessor communication times*. SIAM J. Comput., vol. 18, num. 2, pp. 244-257, April.

[6] Y. K. KWOK AND I. AHMAD (1999). *Static scheduling for task graph allocation*. ACM Computing Surveys, vol. 31, num. 4, December.

[7] R. LEPÈRE AND D. TRYSTRAM (2002). *A new clustering algorithm for scheduling with large communication delays*. 16th IEEE-ACM annual International Parallel and Distributed Processing Symposium (IPDPS'02). April 15-19. Marriott Marina, Fort Lauderdale, Florida

[8] E. HART, P. ROSS, D. CORNE (2005). *Evolutionary scheduling: a review*. Genetic Programming and Evolve Machines, vol. 6, pag. 191-220. 2005 Springer Science + Business Media, Inc. Manufactured in The Netherlands.

[9] A. Y. ZOMAYA AND G. CHAN (2004). *Efficient clustering for parallel tasks execution in distributed systems*. 18th IEEE-ACM annuals International Parallel and Distributed Processing Symposium (IPDPS'04). April 26-30. Santa Fe, New Mexico, Eldorado Hotel.

[10] A. S. WU, H. YU, S. JIN, K. LIN AND G. SCHIAVONE (2004). *An incremental genetic algorithm approach to multiprocessor scheduling*. IEEE Transactions on parallel and distributed systems, vol. 15, num. 9, September.

[11] F. GUINAND, A. MOUKRIM, AND E. SANLAVILLE (2004) *Sensitivity analysis of tree scheduling on two machines with communication delays*. Parallel Computing, 30(1), pages 103-120, 2004.

[12] D. GOLDBERG (1989). *Genetic algorithms in search, optimization and machine learning*. Reading, Mass.: Addison-Wesley.

[13] B. GAUJAL, G. HUARD, E. THIERRY AND D. TRYSTRAM (2004). *Convex Clustering*. 1st Bertinoro Workshop on Algorithms for Scheduling and Communication. 27 june - 3 july 2004. University of Bologna Residential Center. Bertinoro, Italy.

[14] R. C. CORREA, A. FERREIRA, P. REBREYEND (1999). *Scheduling multiprocessor tasks with genetic algorithm*. IEEE Transactions on parallel and distributed systems, vol. 10, num. 8, August.

[15] EMANUEL FALKENAUER (1996). *A hybrid grouping genetic algorithm for bin packing*. Journal of Heuristic, vol. 2, num. 1, pag. 5-30.

[16] Y. KWOK (1994). *Efficient Algorithms for Scheduling and Mapping of Parallel Programs onto Parallel Architectures*. M.Phil. Thesis, Department of Computer Science, The Hong Kong University of Science and Technology, Hong Kong, June, 1994. url = "citeseer.ist.psu.edu/kwok94efficient.html

[17] J.D. EVANS, R.R. KESSLER (1992). *A communication-Ordered Task Graph Allocation Algorithm*. IEEE Transactions on Parallel and Distributed Systems, 1992.

[18] Y.-K. KWOK, I. AHMAD (1999). *Static scheduling algorithms for allocating directed taskgraphs to multiprocessors*. ACM Computing Surveys, vol. 31 num. 4, december, pag. 406-471. 1999.

[19] MIN-YOU WU, WEI SHU, JUN GU (2001). *Efficient Local Search for DAG Scheduling*. IEEE Transactions on Parallel and Distributed Systems, vol. 12 num. 6, june, pag. 617-627. 2001 IEEE Press.

[20] C. L. MCCREARY, M. A. CLEVELAND, AND A. A. KHAN (1996). *The Problem with Critical Path Scheduling Algorithms*. Technical Report 9601, Auburn University. Department of Computer Science and Engineering. January 2, 1996.

Scheduling issues on IBM p690: Performance Analysis with the PARbench Environment

Heiko Dietze[a], Wolfgang E. Nagel[a], Bernd Trenkler[a]

[a]Center for Information Services & High Performance Computing (ZIH) Dresden University of Technology D-01062 Dresden, Germany

1. Introduction

This paper investigates scheduling properties of parallel programs on IBM p690. Based on the PARbench environment, it is the continuation in a series of research studies to analyze different multiprogramming scheduling issues on high performance computer systems ([1], [3], [4], [5]). First, the *PARbench* environment is introduced briefly. Then the performance analysis main results, as the basis for the scheduling study, are described. Finally, the behavior of the scheduling system is discussed.

2. PARbench

The PARbench benchmark system is designed to simulate virtually every workload the user might specify. It can execute many benchmark programs in parallel and record their behavior with regards to time flow. This feature is applied analysing a system workload containing parallel programs in a multiprogramming mode.

The basis for each workload program generated with PARbench is a set of synthetic kernels. All kernels follow two rules: First, they are short enough to combine into one workload. Second, together they are able to represent the overall parameter space of the used computer system.

The used version of PARbench provides 17 different kernels, each consisting of three parts: writing to disk, a loop nest, and reading from disk. The I/O-Part can be varied in 5 stages, including the option to skip it. The loops are often derived from scientific calculations but do not always solve a problem. Every kernel can be used with different data sizes by changing the size of the used matrices. In result, there are 340 different kernel versions available. As a concept for parallelization, OpenMP is used. The number of threads per job during execution is a parameter specified by the user.

3. IBM p690

The tested IBM p690 is a shared memory system with 32 processors. Based on the IBM POWER4+ processor with a clock rate of 1.7 GHz, it has a theoretical peak performance of 6.8 GFLOPS per processor.

3.1. Preliminary Investigations

The first step for the work with the PARbench benchmark is to identify the kernel properties. This includes, amongst other things, the FLOP-rate and rate of the processors memory references per second (MREFS). The results represent the available parameter space for the scheduling experiments, particularly for the CPU and memory utilization.

The review of the results for all 340 Kernels shows that very few kernels reach a rate of more than 25% theoretical peak performance (1700 MFLOPS) and yet fewer reach more than 50% theoretical

peak performance (3400 MFLOPS). The same, but to a lesser degree, can be said about the rate of memory references. A plot of all kernels is available in Figure 1. Importantly, a high FLOP-rate cannot be achieved in conjunction with a high rate of memory references. This behavior is due mainly to the processor and memory interface, but the PARbench benchmark properties can be of influence.

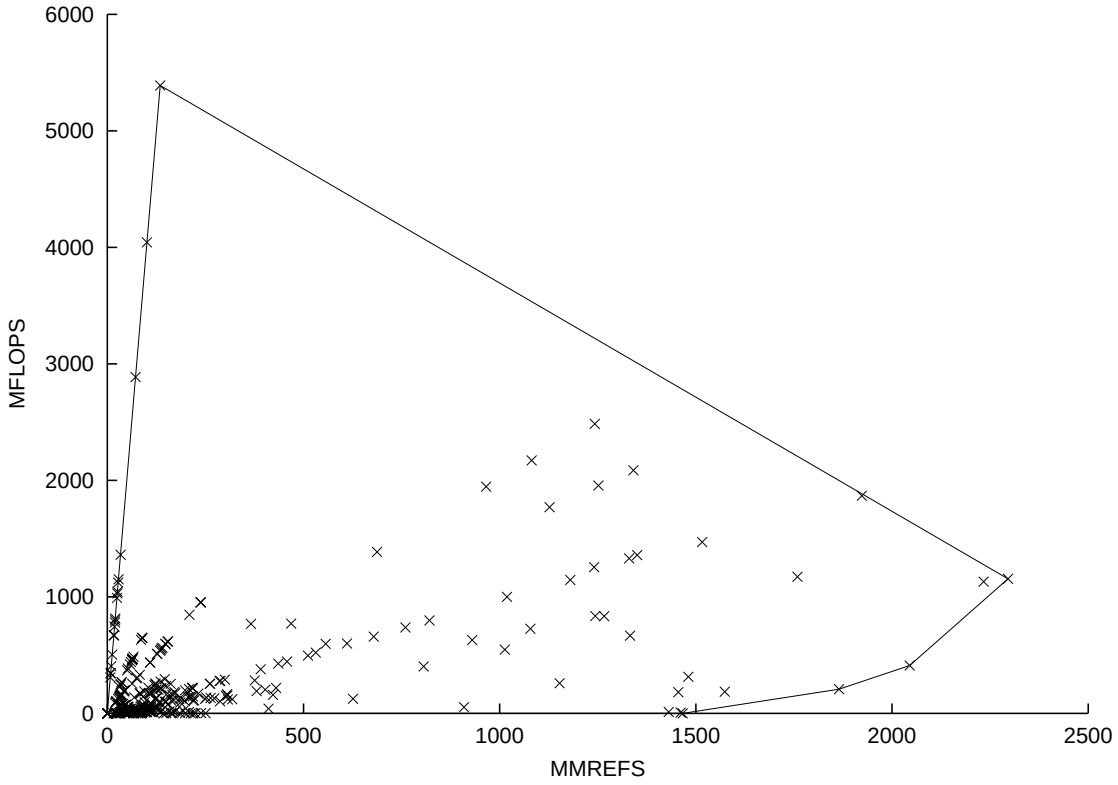


Figure 1. PARbench kernel performance (MMREFS and MFLOPS) on IBM p690

To reach the theoretical floating-point peak performance, it is necessary to use both floating-point-units continuously with FMA-Operations (Fused Multiply Add Operation). This is not possible in all PARbench-Kernels. This disables the attainable rate down to 1.7 GFLOPS. Even if it was possible, insufficient bandwidth and high latencies to the memory and level-three (L3) cache prevent a better performance. The design of the processor core also reduces the attainable FLOP-rate. Managing out-of-order execution via groups is a design decisions. Tracking by groups of instructions and not individual instructions reduces the necessary hardware, but introduces additional overhead such as the cost of creating an instruction group and the cost due to lost opportunities for parallel computation.

Another problematic design feature is the two floating-point-units. Each unit has a six staged pipeline. However, each pipeline stage needs an independent floating-point instruction to operate continuously. Thus, 12 or more instructions are needed to feed the pipeline in an continuous and efficient way. This can be a problematic requirement for loops with data dependencies (even in one iteration). The peak performance of 5.4 GFLOPS (Kernel number 1) can only be achieved by using 40 FMA-register-register-operations per iteration of the inner loop.

Another aspect is the processors memory interface. There are two load/store-units per processor core. Every unit can issue one load or store operation per clock cycle. Thus the theoretical limit for the memory references is 3800 million (3800 MMREFS). Consequently, each floating-point-unit is only provided with one operand from memory (or caches) per cycle.

The results indicate that the memory subsystem is a bottleneck of the system. The high latency of the level-three (L3) cache and structural problem of the shared level-two (L2) cache increase this problem. Thus, only 2.3 GMREFS (Peak 3.8 GMREFS) could be achieved with PARbench kernel number 186. This kernel uses the level-one (L1) data-cache very efficiently, because it uses 15 out of 16 entries of a cache line and the hardware pre-fetching can be employed.

4. Scheduling on the IBM p690

Scheduling is the task of the operating system. The tested IBM p690 system is equipped with the AIX 5L Version 5.2 for POWER. It uses a thread based scheduling system with priority queues (256 stages). The scheduling algorithm is a fair round robin algorithm with dynamic priorities. The priorities are calculated on the threads CPU usage. Since AIX 4.3.3, each processor has its own queue (32 queues for 32 processors). Furthermore, there is one global queue for all processors available. However, this queue must be explicitly activated and overwrites the system of local queues.

4.1. Serial Workloads

The initial experiments investigate the basic scheduling behavior. This includes workloads based on kernel number 1 and 186. The jobs based on kernel 1 represent the ideal workload without side effects. A run with 32 identical jobs based on kernel 1 and with 32 processors is nearly perfect. All 32 jobs run without any waiting time. There is virtually no additional user-time and the system-time, an indicator for the schedulings overhead, is negligible.

The same experiment, with the 32 identical jobs, based on the memory intensive kernel 186 results differently. There is a visible prolongation of the user-time for several jobs. This is explained by the interaction of the jobs when caches are shared. When caches are shared, more time is needed during loading and storing.

Both experiments show that scheduling serial workloads in full-load situations are unproblematic. However, cache interference and memory must be watched, as it can increase the necessary user-time.

4.2. Serial Workloads with Overload

The experiments in overload situations are simulated by using 48 jobs based on kernel 1. Each job has a initial runtime of 100 seconds. The operating system offers two variants for organizing the scheduling: one queue per processor, which is the standard version (results shown in Figure 2), or one global queue for all processors (not shown). The scheduling algorithm for all queues is the fair round robin system with dynamic priorities.

The results from the system of 32 queues show two classes of jobs: with and without waiting time. Naturally, slight waiting time due to the overload is to be expected. The reason for the difference is the late migration between the processor queues. If a processor is idle, one of the waiting jobs migrates to the other processor. In the beginning of the experiment with 48 jobs there are queues with only one job and queues with two jobs. The single jobs run without interruptions until completion. Meanwhile, two jobs share one processor fairly via round robin. Only after completion of one quasi exclusive jobs is there a migration (late migration). Now the two jobs can complete without interruption.

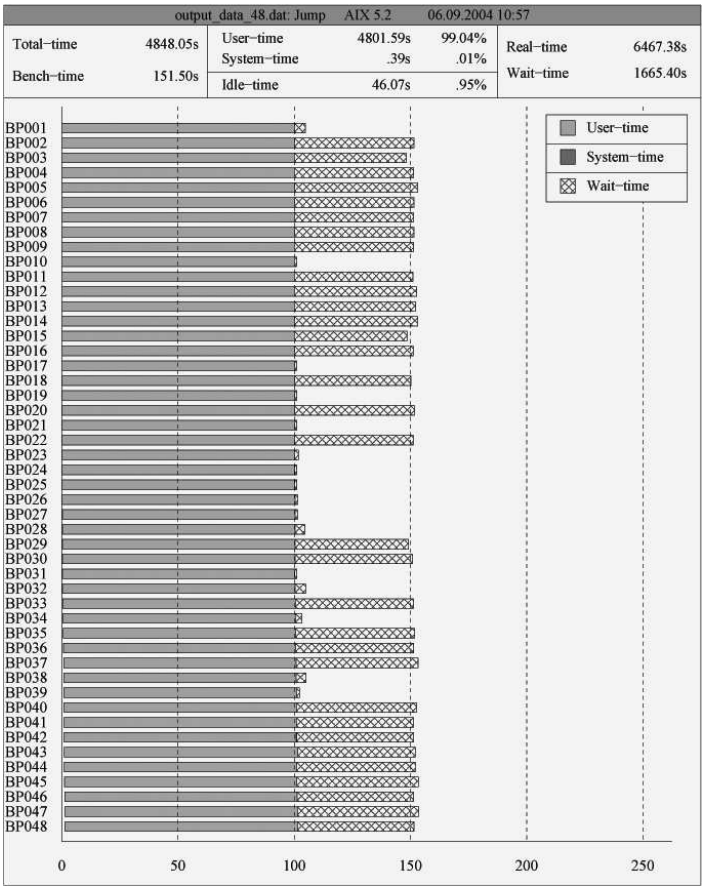


Figure 2. PARbench experiment with 48 serial jobs based on kernel 1 running on 32 processors with one queue per processor.

Using the global queue for scheduling, the result is quite different. All jobs are distributed equally among the 32 processors by the round robin scheduling algorithm. This means every job has now a waiting time of approximately half the runtime. Due to the loss of several processors exclusive use, the overall waiting time increases. The global queue in conjunction with a fair scheduling concludes in an increase of waiting time.

4.3. Mixed Workloads

For experiments with parallel workloads, we create a new set of 32 jobs. Every job is created with the goal of 1900 MFLOPS, 1000 MMREFS and 100 seconds runtime. This is impossible with one special kernel, so a sequence of kernels are used. This sequence contains different kernels to approximate the given job properties.

As an example of a mixed workload with both parallelized and serial jobs, we use 16 serial jobs and 16 parallel jobs with 4 threads each. The jobs are divided into four groups, each with 4 serial and 4 parallel jobs. The result for the system of one queue per processor is shown in Figure 3 and for the global queue in Figure 4.

The system of local queues produce only a small speedup, but this is done at a high cost. The overall user-time increases and the waiting time is not reduced. Therefore, there is no actual gain in runtime. This is due to the migration between the 32 queues of the 32 processors.

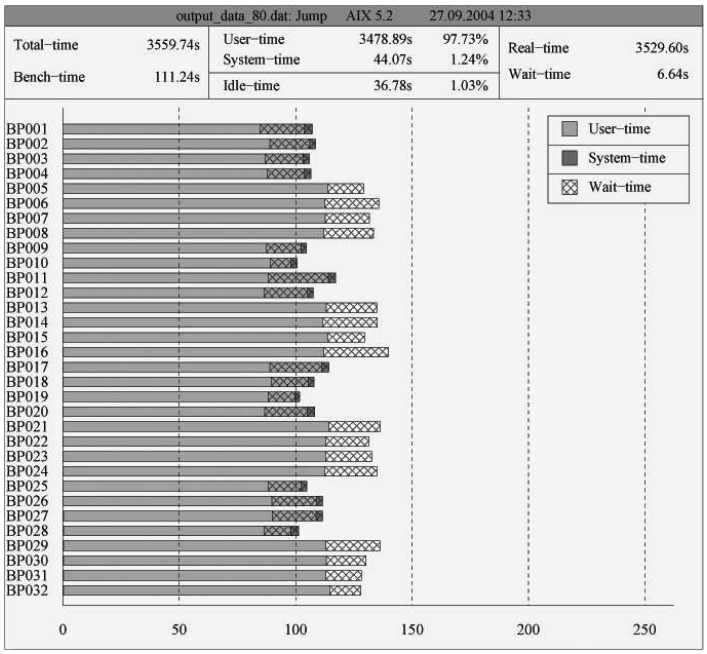


Figure 3. PARbench experiment with 16 jobs a 4 threads and 16 serial jobs running on 32 processors with one queue per processor.

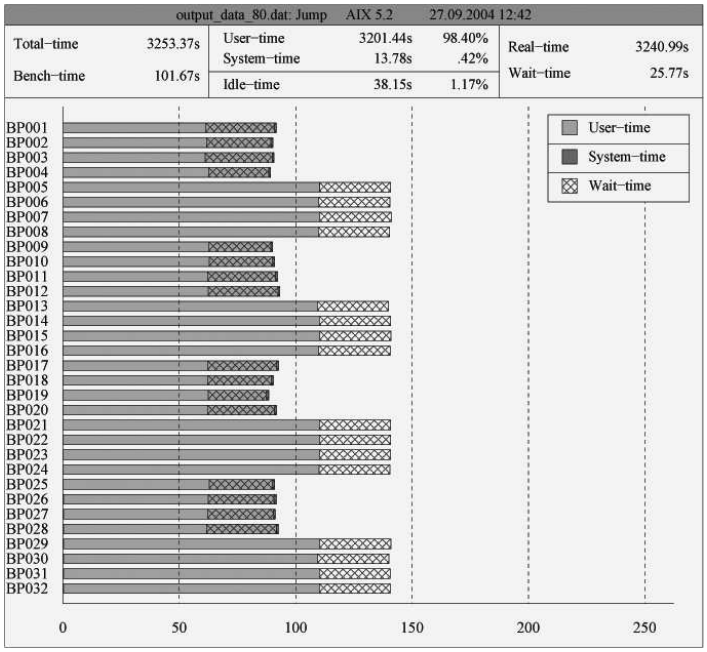


Figure 4. PARbench experiment with 16 jobs a 4 threads and 16 serial jobs running on 32 processors with the global queue.

With initiation, every job is assigned to a queue and processor. Then, the 16 jobs start to execute their parallel regions (OpenMP). Still, the 4 threads of one parallel job remain mainly on the same processor. The late migration reduces the ability to work with threads of the same job in parallel. A consequence of the increased busy-wait is additional user-time.

The results from the global queue (Figure 4) show a better acceleration than the local queues. Also, there is no additional overall user-time. However, due to the properties of the global queue, there is an increase of the overall waiting time. This annuls any gain due to parallelization in comparison to the serial working of the set of jobs. The two possibilities for organizing scheduling do not produce convincing results in an overload situation with mixed workloads of parallel and serial jobs.

During the experiments with mixed workloads, we encountered some shortcomings in the accounting system, particularly with the user-time. In result, the accounting system recorded less user-time for parallel programs (OpenMP) and thus more to the sequential programs. In extreme cases, we measured sequential programs with a higher user-time than real-time. Yet, the sum of all user-times remains correct. This seems to occur only if thread-parallel and sequential programs shared the same queue. Thus, the drawn user-time in Figures 3 and 4 for parallel jobs is, in reality, higher and for sequential jobs, smaller. With parallel-only or sequential workloads, there are no such visible effects.

The experiment with a parallel-only workload uses a set of 32 parallel jobs with 4 threads per job. The results with local queues are shown in Figure 5. There are no gains visible and the user-time

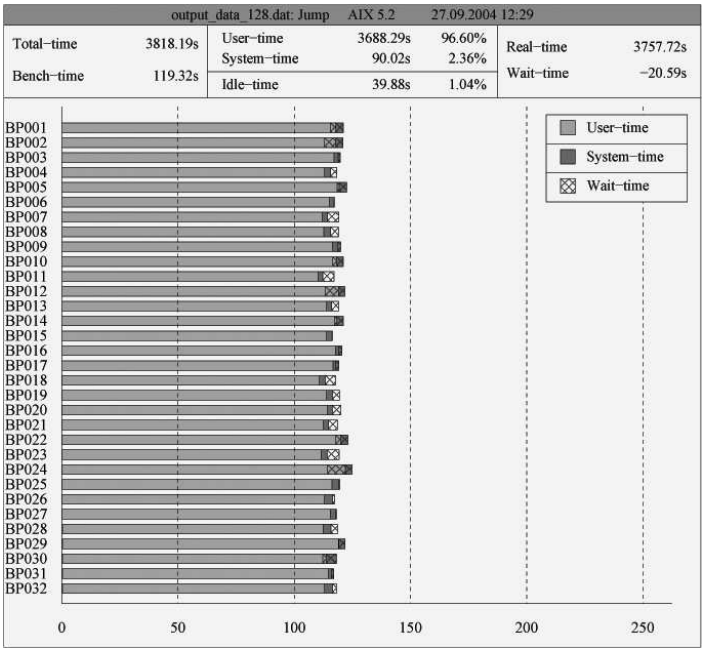


Figure 5. PARbench experiment with 32 jobs a 4 threads with one queue per processor.

increases even further. The effect of one queue per processor is most visible here: no thread has a chance to work in parallel with a thread of its own job because no thread can migrate to another processor due to late migration. The increase of user-time is explained due to the busy-wait during parallelization.

Changing to the global queue (not shown) could stop the additional user-time, but there is still no gain through parallelization. Serializing the threads is nearly ideal and this case is viewed as almost identical to the calculation with 32 serial jobs. The only cost is the additional work for the scheduler, which can be measured in an increase of the system-time.

4.4. Handoptimized Workload

The main goal of parallel work is to gain an acceleration due to reduced runtime. The current results show that a system of local queues have a negative influence on parallel computation. The global queue is slightly better but it also increases the overall waiting time thus negating any positive effects.

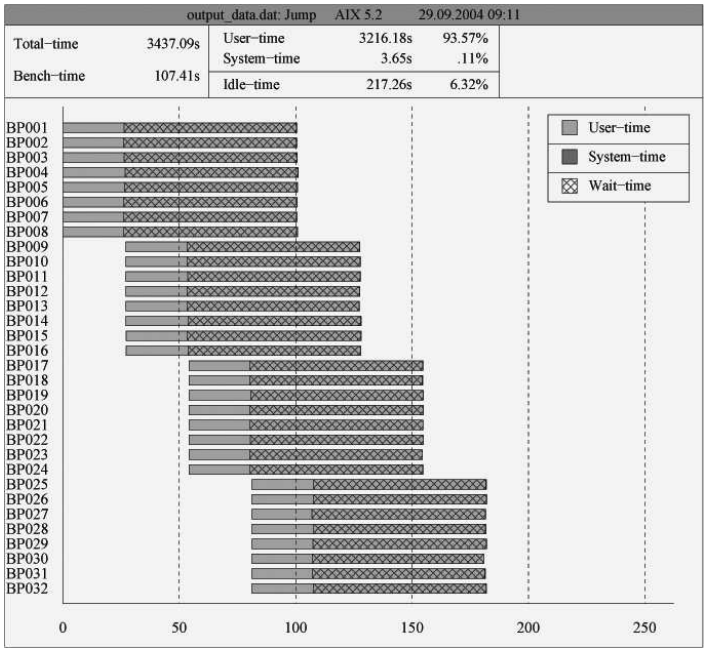


Figure 6. PARbench experiment with 32 jobs a 4 threads and hand-optimization delay in the beginning of parts of the workload.

To avoid the overloads negative effects, one can use a delay when starting some of the jobs in the workload. This is done, for instance, manually, as shown with PARbench in Figure 6 (for the example of 32 jobs with 4 threads per job).

This results in nearly ideal behavior. Every job can gain a considerable acceleration and nearly all jobs finish earlier than in a sequential mode. Only the last group of 4 jobs completes after the 100 second limit. The reason being the acceleration is slightly less than ideal for a parallelization with 4 processors. But this is negligible in contrast to the results without a manual optimization (ref. Figure 5).

A manual optimization or the use of a workload manager to prevent an overload situation contradicts the ideal multiprogramming mode. Additionally, the use of MPI in a cluster of IBM p690 nodes is only efficient if all processes of one MPI-job are active at the same time. This can not be guaranteed in an overload situation with multiprogramming.

5. Summary and Outlook

The performance results show that the POWER4+ Processor has several deficits reaching its peak performance. The main reasons are the design of the processor core and the memory-interface. The caches can not compensate this problem. The level-three (L3) cache with its high latency and the shared level-two (L2) cache are also problematic factors.

The operating systems scheduling system works perfect for workloads up to a load of 100 percent. In overload situations, such as multiprogramming environments, both scheduling variants exhibit drawbacks. Using the global queue can produce acceptable results but at the cost of higher waiting times. The method of one queue per processor produces the best results with serial workloads. Its disadvantage being the longer process-time when used in conjunction with parallel jobs. Additionally, the speedup is small or the parallel runtime is longer than the sequential version. A manual optimization using a start-delay to decrease the workload could produce the best results. Furthermore, we have detected some shortcomings in the accounting system which, just by accident, record fewer to the parallel programs.

In the future, limited degree of parallelism will be tested. It is expected, that multiprogramming provides better results in this situation, as it is the mostly used dedicated mode. Furthermore, other machines e.g. Sun Fire E25K, NEC-SX8 will be tested.

6. Acknowledgment

We would like to thank the John-von-Neuman Institute at the Research Center Juelich for providing the access to their IBM p690 Cluster JUMP thus making the measurements possible.

References

[1] Sebastian Boesler. Performance-Analyse von Hochleistungsrechnern im Multiprogramming-Betrieb: Untersuchungen auf der SGI Origin. Diploma-Thesis, Dresden University, Dec 2001.

[2] Heiko Dietze. Das PARbench-System: Untersuchungen zum Scheduling von parallelen Programmen auf der IBM p690. Diploma-Thesis, Dresden University, Nov 2004.

[3] Klaus Fabian. Leistungsuntersuchungen von Multiprozessorsystemen auf der Basis des parametergesteuerten Lastbeschreibungssystems PAR-Bench unter besonderer Bercksichtigung von parallel ablaufbaren Teillasten. Technical Report Jl-2671, Forschungszentrum Jlich, August 1992.

[4] Andreas Kowarz. Performance-Untersuchungen mit dem PARbench-System auf unterschiedlichen Parallelrechnern. Diplomarbeit, Technische Universitt Dresden, Fakultt Informatik, Institut fr Technische Informatik, Mai 2003.

[5] Wolfgang E. Nagel and Markus A. Linn. Benchmarking parallel programs in a muliprogramming enviroment: The PARbench system, 1991.

Minisymposium

Bioinformatics

Implementation of Anisotropic Nonlinear Diffusion for filtering 3D images in Structural Biology on SMP Clusters*

Siham Tabik[a], E. M. Garzón[a], I. García[a], J.J. Fernández[a]

[a]Dept. de Arquitectura de Computadores y Electrónica. Universidad de Almería. 04120 Almería

In this work we have proposed a parallel implementation of the Anisotropic Nonlinear Diffusion (AND) for filtering 3D images. AND is a powerful noise reduction technique in the field of computer vision. This method is based on a partial differential equation (PDE) tightly coupled with a massive set of eigensystems. Denoising large (3D) images in biomedicine and structural cellular biology by AND has a high computational cost. Consequently, an appropriate parallel implementation of AND is the best approach to reduce its runtime. This work proposes a portable parallel code using several programming models:(1) shared address space model, (2) message passing model and (3) hybrid model. The proposed parallel implementation has been evaluated on a cluster of biprocessors. The evaluation of the performance of AND-code shows that the hybrid model scales better in this kind of platforms.

1. Introduction

In many disciplines, raw data acquired from instruments are substantially corrupted by noise and sophisticated filtering techniques are then indispensable for a proper interpretation or post-processing. In general terms, smoothing techniques can be classified into linear and non-linear. Standard linear filtering techniques based on local averages or Gaussian kernels succeed in reducing the noise, but at expenses of poor feature preservation. In other words, they may severely blur the features as their edges are attenuated. However, nonlinear filtering techniques achieve better feature preservation as they try to adaptively tune the strength of the smoothing to the local structures found in the image.

Anisotropic nonlinear diffusion (AND) is currently one of the most powerful noise reduction techniques in the field of computer vision [20]. This technique takes into account the local structures found in the image to filter noise, preserve edges and enhance some features, thus considerably increasing the signal-to-noise ratio (SNR) with no significant quantitative distortions of the signal. Pioneered in 1990 by Perona and Malik [17], AND has grown to become a well-established tool in the last decade [15,20–22]. AND has already been successfully applied in different disciplines, such as medicine [2,11,12] or biology [8–10], for denoising multidimensional images. AND has actually been crucial to achieve some recent breakthroughs [4,6,13,16].

The mathematical basis of AND is a partial differential equation (PDE) tightly coupled with a massive set of eigensystems [18]. AND may turn out to be extremely expensive, from the computational point of view, depending on the size of the images. There are some disciplines where the requirements may be so huge –much more than 1 Gbyte in size [7]– that parallel computing proves to be essential.

The standard numerical scheme for solving PDEs is based upon an explicit finite difference discretization. More efficient schemes have been specifically designed for nonlinear diffusion [23], though. However, they are complex to implement and, despite their efficiency, they still require to be parallelized [5].

*This work was supported by the Spanish Ministry of Education and Science through grant TIC2002-00228

In this work we address the parallelization of AND for its application to denoising of large three-dimensional (3D) volumes in biomedicine and structural cellular biology. We make use of the standard explicit numerical scheme for the discretization. This scheme is commonly used in other fields where PDEs are involved [18] and, as a consequence, the parallel approaches that are presented and discussed here may be valuable for them too.

2. Review of anisotropic nonlinear diffusion

AND accomplishes a sophisticated edge-preserving denoising that takes into account the structures at local scales. AND tunes the strength of the smoothing along different directions based on the local structure estimated at every point of the multidimensional image. Conceptually speaking, AND can be considered as an adaptive gaussian filtering technique in which, for every voxel in the volume, an anisotropic 3D gaussian function is computed whose widths and orientations depend on the local structure [3]. This section presents local structure determination via structure tensors, the concept of diffusion, a diffusion approach commonly used in image processing and, finally, details of the numerical implementation.

2.1. Estimation of local structure

The *structure tensor* is the mathematical tool that allows us to estimate the local structure in a multidimensional image. Let $I(\mathbf{x})$ denote a 3D image, where $\mathbf{x} = (x, y, z)$ is the coordinate vector. The structure tensor of I is a symmetric positive semi-definite matrix given by:

$$\mathbf{J}(\nabla I) = \nabla I \cdot \nabla I^T = \begin{bmatrix} I_x^2 & I_x I_y & I_x I_z \\ I_x I_y & I_y^2 & I_y I_z \\ I_x I_z & I_y I_z & I_z^2 \end{bmatrix} \tag{1}$$

where $I_x = \frac{\partial I}{\partial x}$, $I_y = \frac{\partial I}{\partial y}$, $I_z = \frac{\partial I}{\partial z}$ are the derivatives of the image with respect to x, y and z, respectively.

The eigen-analysis of the structure tensor allows determination of the local structural features in the image [20]:

$$\mathbf{J}(\nabla I) = [\mathbf{v_1}\ \mathbf{v_2}\ \mathbf{v_3}] \cdot \begin{bmatrix} \mu_1 & 0 & 0 \\ 0 & \mu_2 & 0 \\ 0 & 0 & \mu_3 \end{bmatrix} \cdot [\mathbf{v_1}\ \mathbf{v_2}\ \mathbf{v_3}]^T \tag{2}$$

The orthogonal eigenvectors $\mathbf{v_1}$, $\mathbf{v_2}$, $\mathbf{v_3}$ provide the preferred local orientations, and the corresponding eigenvalues μ_1, μ_2, μ_3 (assume $\mu_1 \geq \mu_2 \geq \mu_3$) provide the average contrast along these directions. The first eigenvector $\mathbf{v_1}$ represents the direction of the maximum variance. Therefore, $\mathbf{v_1}$ represents the direction normal to the local feature (see Fig. 1).

2.2. Concept of diffusion in image processing

Diffusion is a physical process that equilibrates concentration differences as a function of time, without creating or destroying mass. In image processing, density values play the role of concentration. This observation is expressed by the *diffusion equation* [20]:

$$I_t = \mathrm{div}(\mathbf{D} \cdot \nabla I) \tag{3}$$

where $I_t = \frac{\partial I}{\partial t}$ denotes the derivative of the image I with respect to the time t, ∇I is the gradient vector, $\mathbf{D}$ is a square matrix called *diffusion tensor* and div is the *divergence* operator:

$$\mathrm{div}(\mathbf{f}) = \frac{\partial f_x}{\partial x} + \frac{\partial f_y}{\partial y} + \frac{\partial f_z}{\partial z}$$

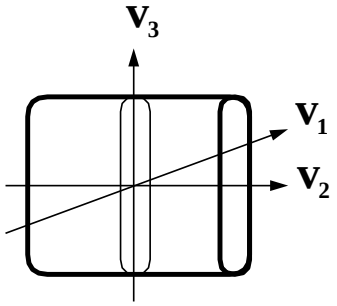


Figure 1. Local structure found by eigen-analysis of the structure tensor. $\mathbf{v_1}$, $\mathbf{v_2}$, $\mathbf{v_3}$ are the corresponding eigenvectors. $\mathbf{v_1}$ is the direction normal to the local structure.

In AND the smoothing depends on both the strength of the gradient and its direction measured at a local scale. The diffusion tensor $\mathbf{D}$ is therefore defined as a function of the structure tensor J:

$$\mathbf{D} = [\mathbf{v_1}\ \mathbf{v_2}\ \mathbf{v_3}] \cdot \begin{bmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & \lambda_3 \end{bmatrix} \cdot [\mathbf{v_1}\ \mathbf{v_2}\ \mathbf{v_3}]^T \tag{4}$$

where $\mathbf{v_i}$ denotes the eigenvectors of the structure tensor. The values of the eigenvalues λ_i define the strength of the smoothing along the direction of the corresponding eigenvector $\mathbf{v_i}$. The values of λ_i rank from 0 (no smoothing) to 1 (strong smoothing).

Therefore, this approach allows smoothing to take place anisotropically according to the eigenvectors determined from the local structure of the image. Consequently, AND allows smoothing on the edges: Smoothing runs along the edges so that they are not only preserved but smoothed. AND has turned out, by far, the most effective denoising method by its capabilities for structure preservation and feature enhancement [8,9,20].

2.3. Edge Enhancing Diffusion

One of the most common ways of setting up the diffusion tensor $\mathbf{D}$ gives rise to the so-called Edge Enhancing Diffusion (EED) approach [20]. The primary effects of EED are edge preservation and enhancement. Here strong smoothing is applied along the preferred directions of the local structure, (the second and third eigenvectors, $\mathbf{v_2}$ and $\mathbf{v_3}$). The strength of the smoothing along the normal of the structure, i.e. the eigenvector $\mathbf{v_1}$, depends on the gradient: the higher the value is, the lower the smoothing strength is. Consequently, λ_i are then set up as:

$$\begin{cases} \lambda_1 = g(|\nabla I|) \\ \lambda_2 = 1 \\ \lambda_3 = 1 \end{cases} \tag{5}$$

with g being a monotonically decreasing function, such as $g(x) = 1 - \exp\left(\frac{-3.31488}{(x/K)^8}\right)$, where $K > 0$ acts as a contrast parameter [20]; Structures with $|\nabla I| > K$ are regarded as edges, otherwise they are considered to belong to the interior of a region. Therefore, smoothing along edges is preferred over smoothing across them, hence edges are preserved and enhanced.

2.4. Numerical discretization of the diffusion equation

The diffusion equation, Eq. (3), can be numerically solved using finite differences. The term $I_t = \frac{\partial I}{\partial t}$ can be replaced by an Euler forward difference approximation. The resulting explicit

scheme allows calculation of subsequent versions of the image iteratively:

$$I^{(s)} = I^{(s-1)} + \tau \cdot (\frac{\partial}{\partial x}(D_{11}I_x) + \frac{\partial}{\partial x}(D_{12}I_y) + \frac{\partial}{\partial x}(D_{13}I_z) + \frac{\partial}{\partial y}(D_{21}I_x) + \frac{\partial}{\partial y}(D_{22}I_y) + \frac{\partial}{\partial y}(D_{23}I_z) + \frac{\partial}{\partial z}(D_{31}I_x) + \frac{\partial}{\partial z}(D_{32}I_y) + \frac{\partial}{\partial z}(D_{33}I_z)) \quad (6)$$

where τ denotes the time step size, $I^{(s)}$ denotes the image at time $t_s = s\tau$, the terms I_x, I_y, I_z are the derivatives of the image with respect to x, y and z, respectively. Finally, the D_{mn} terms represent the components of the diffusion tensor $\mathbf{D}$. The standard scheme to approximate the spatial derivatives ($\frac{\partial}{\partial x}$, $\frac{\partial}{\partial y}$ and $\frac{\partial}{\partial z}$) is based on central differences.

In this traditional explicit scheme for solving the partial differential equation Eq. (3), the stability is an issue [20]. The maximum time step that is allowed is $\tau \leq 0.5/N_d$, where N_d is the number of dimensions of the problem. In our case, we are dealing with a three-dimensional problem, so $N_d = 3$. In the experiments carried out in this work, we used a conservative value of $\tau = 0.1$.

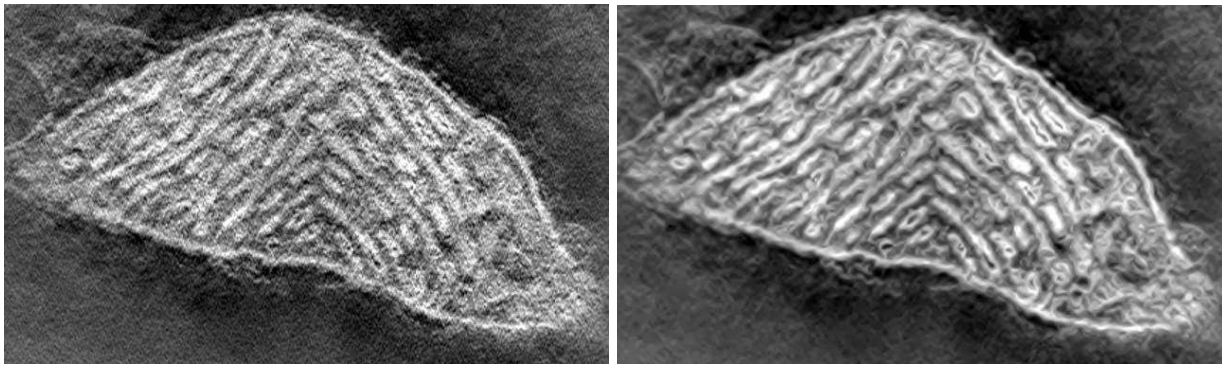


Figure 2. Left: slice from the original volume of a mitochondrion; right: slice from the filtered volume

Fig. 2 shows the result of the application of AND to a volume of a mitochondrion, a cell organelle. The enhancement in visualizing a slice of the volume is apparent (up: slice from the original volume; bottom: slice from the filtered volume).

2.5. Scheme of the diffusion approach

The outline of the AND approach is the following:

(1) Compute the image at iteration s. For every voxel:
  Compute the structure tensor $\mathbf{J}$ according to Eqs.(1) and (2).
  Compute the diffusion tensor $\mathbf{D}$ according to Eq.(4).
  Solve the partial differential equation of diffusion, Eq. (3).
(2) Iterate: go to step (1)

3. Parallel implementation of AND

Large scale parallel systems based on clusters of shared memory symmetric multiprocessors (SMP) are today's dominant computing platforms, where the communication between nodes is established by means of message passing across the network and inside the nodes the communication

between processors is established by means of the shared memory [14,19]. This architecture supports three parallel programming models: (1) shared address space model, (2) message passing model and (3) hybrid model. The hybrid model combines the two previous models, shared address space model inside nodes and message passing model between nodes.

In this work a portable code has been designed based on a hybrid model, which can be translated into shared address space and/or message passing models, taking into account the features of the considered parallel platform. The hybrid implementation of the AND-code uses two standards: MPI for the message passing between nodes and Pthreads for the shared memory parallelism inside the nodes.

The hybrid implementation is based on two features of AND:

- At each time step, s, the 3D image is updated by z-planes, where the z-axis is the direction of the larger image dimension N_z.
- The update of the k z-plane, I_k^{s+1}, is only function of I_k^s; and its four neighbour z-planes, I_{k-1}^s, I_{k-2}^s, I_{k+1}^s and I_{k+2}^s, where s denotes the iteration index and k denotes the z-plane index.

To optimize data access and storage of the AND-code, memory requirements have been minimized, allocating and computing only the necessary data needed for updating each single z-plane. The following parallel algorithm takes into account the underlined scheme:

Distribute I^0 among nodes, N_z^{local} planes will be assigned to each node
Do $s = 1, ...n$
  (a) Each thread initializes its auxiliary data structures
  (b) Do $k = 2, ..., \lceil (N_z^{local} - 4)/T \rceil$
    $I_k^{s+1} = AND(I_k^s)$
  End Do
  (c) Interchange boundary z-planes between neighbour nodes.
  End Do
Collect the image.

where N_z^{local} denotes the local number of z-planes of the image in every node, T denotes the number of processors inside one node, and I^0 denotes the original 3D image, and n denotes the number of iterations to obtain an acceptable denoised image. Dimensions of the volumes in structural biology usually range between 256x256x256 and 768x768x768. Typical values for n are around 60-100 iterations

It was necessary to control the data distribution at two levels, at node level and at processor level inside the node:

1. The total number of z-planes is distributed among nodes, $N_z^{local} = \lceil N_z/P \rceil + 4$ z-planes will be assigned to each node, where P denotes the number of nodes. To decrease message passing communications at each time step, for updating boundary planes of the diffusion tensor all nodes allocate four additional planes to hold four neighbour planes: two planes from each boundary, as it is illustrated in Figure 3. At the end of each time step, nodes communicate the updated four boundary planes via message passing.
2. At processor level, each thread will update its subset of $\lceil (N_z^{local} - 4)/T \rceil$ z-planes applying the AND process, where a z-plane of the structure tensor and of the diffusion tensor are computed

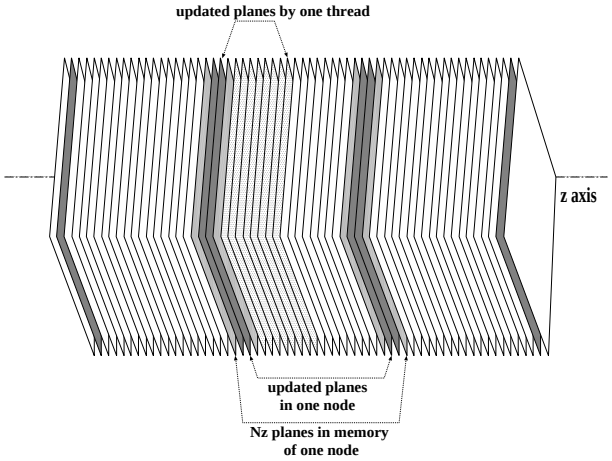


Figure 3. The distribution of an image at node level and processor level.

to update the corresponding z-plane of the image. To carry out this process, two auxiliary data structures have been defined by each thread, in order to hold both the boundary z-plane of the image and the boundary z-plane of the diffusion tensor, before they are modified by the thread neighbour.

In the next section, the described parallel code will be evaluated on a cluster of SMP.

4. Evaluation of the Parallel implementation of AND

In this paper, we evaluate the performance of a portable implementation of AND-code as message passing code and as hybrid code on a cluster of SMP multiprocessors. The aim of this evaluation is to determine which model exploits best the SMP clusters.

The implementation is carried out on a cluster of biprocessors of Intel(R) Xeon(TM) 3.06 GHz with 2 GB RAM, 512 KB cache. Nodes are interconnected via two Gigabit Ethernet networks, one for data (NFS) and the other for computation. Four test volumes of different sizes (256x256x256, 384x384x384, 512x512x512 and 640x640x640) have been selected to develop the evaluation process.

Next, the speed-up will be analyzed for the test volumes. Traditionally, the speedup is only referred to the sequential runtime. Recently, a general concept of speedup has been introduced [1], where the parallel runtime is used as reference instead. In this evaluation, this concept is tacked into account to evaluate the performance of AND on the cluster of SMPs. Specifically, for the volumes 512x512x512 and 640x640x640 the parallel run time with four processors is considered as reference, since it was not possible to run the codes on fewer than four processors with these volumes. While the sequential times is used as reference for the smaller volumes 256x256x256 and 384x384x384.

Fig 4 shows the speedup achieved by the pure MPI code and the hybrid code, for the test volumes, on the cluster of SMPs described above. In global terms, both strategies yield good results, with slightly better performance for the hybrid strategy. It is evident from these figures that the hybrid strategy yields better scalability than the strategy based on message passing, specially for increasing number of processors. Moreover, the influence of the volume size has proven to be relevant on this platform, given that the speedup is better for large sizes.

Finally, notice that $N_z = N_x = N_y$ for the test images considered in this work; better performance is expected if the z-dimension of the image verifies: $N_z >> N_x,\ N_y$.

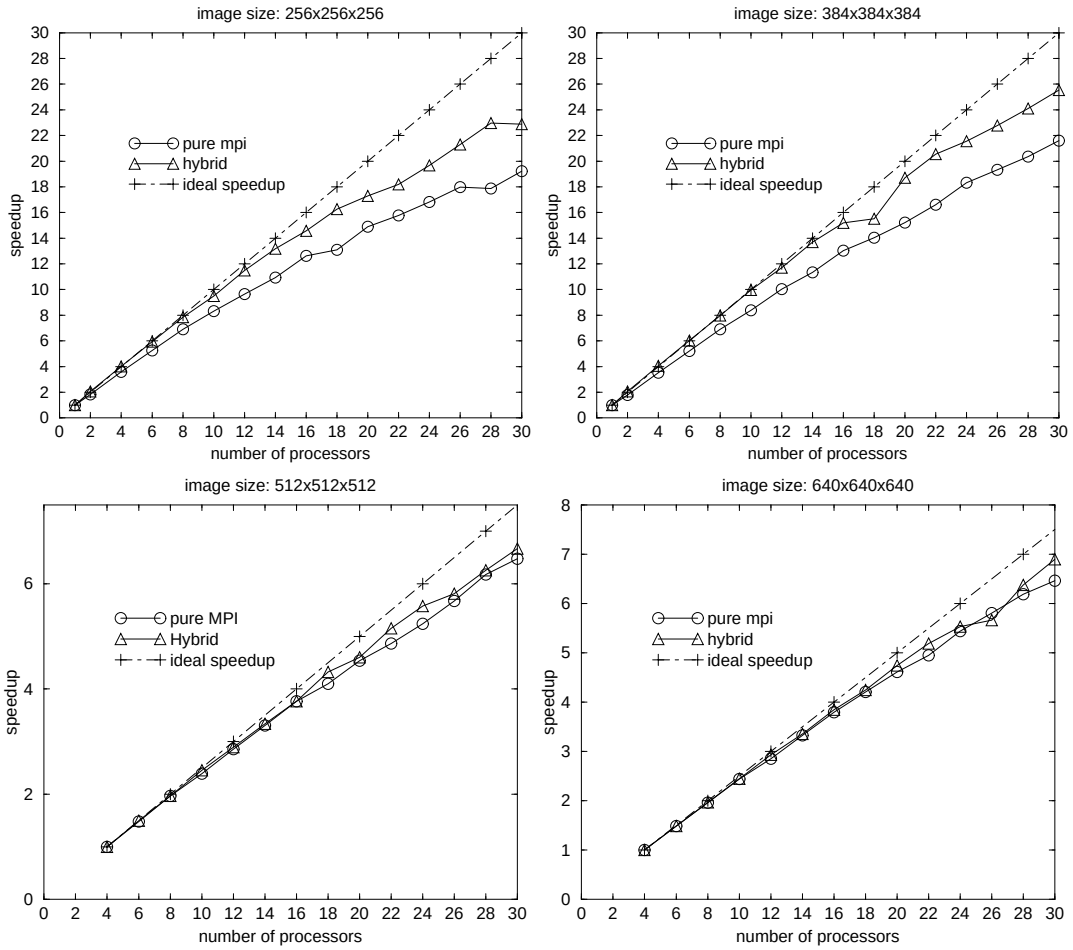


Figure 4. Speed-up of the message passing and the hybrid models versus the number of processors for the test volumes.

5. Conclusions and future works

It can be concluded from the evaluation that the hybrid implementation of the AND-code allows: (1) to reduce significantly the runtime over a large range of images size, specially for images of large sizes and (2) to be run on different kind of parallel platforms. The evaluation shows that the parallel AND-code is scalable, as well as the hybrid model achieves better performance in SMP clusters. In future works, an evaluation of the AND-code will be developed in an extended range of platforms.

Acknowledgments

The authors wish to thank Dr. Guy Perkins (National Center for Microscopy and Imaging Research, San Diego, USA) for kindly providing the mitochondrion dataset.

References

[1] S. G. Akl. Superlinear performance in real-time parallel computation. *The Journal of Supercomputing*, 29(1):89–111, 2004.

[2] I. Bajla and I. Hollander. Nonlinear filtering of magnetic resonance tomograms by geometry-driven diffusion. *Machine Vision and Applications*, 10:243–255, 1998.

[3] D. Barash. A fundamental relationship between bilateral filtering, adaptive smoothing and the nonlinear diffusion equation. *IEEE Trans. Patt. Anal. Mach. Intel.*, 24:844–847, 2002.

[4] M. Beck, F. Forster, M. Ecke, J. M. Plitzko, F. Melchior, G. Gerisch, W. Baumeister, and O. Medalia. Nuclear pore complex structure and dynamics revealed by cryoelectron tomography. *Science*, 306:1387–1390, 2004.

[5] A. Bruhn, T. Jakob, M. Fischer, T. Kohlberger, J. Weickert, U. Bruning, and C. Schnorr. High performance cluster computing with 3-D nonlinear diffusion filters. *Real-Time Imaging*, 10:41–51, 2004.

[6] M. Cyrklaff, C. Risco, J. J. Fernandez, M. V. Jimenez, M. Esteban, W. Baumeister, and J. L. Carrascosa. Cryo-electron tomography of vaccinia virus. *Proc. Natl. Acad. Sci. USA*, 102:2772–2777, 2005.

[7] J. J. Fernandez, J.M. Carazo, and I. Garcia. Three-dimensional reconstruction of cellular structures by electron microscope tomography and parallel computing. *J. Paral. Distr. Computing*, 64:285–300, 2004.

[8] J. J. Fernandez and S. Li. An improved algorithm for anisotropic nonlinear diffusion for denoising cryo-tomograms. *J. Struct. Biol.*, 144:152–161, 2003.

[9] A. S. Frangakis and R. Hegerl. Noise reduction in electron tomographic reconstructions using nonlinear anisotropic diffusion. *J. Struct. Biol.*, 135:239–250, 2001.

[10] A. S. Frangakis, A. Stoschek, and R. Hegerl. Wavelet transform filtering and nonlinear anisotropic diffusion assessed for signal reconstruction performance on multidimensional biomedical data. *IEEE Trans. BioMed. Engineering.*, 48:213–222, 2001.

[11] G. Gerig, R. Kikinis, O. Kubler, and F. A. Jolesz. Nonlinear anisotropic filtering of MRI data. *IEEE Trans. Med. Imaging*, 11:221–232, 1992.

[12] O. Ghita, K. Robinson, M. Lynch, and P. F. Whelan. MRI diffusion-based filtering: A note on performance characterisation. *Computerized Medical Imaging and Graphics*, 29:267–277, 2005.

[13] K. Grunewald, P. Desai, D. C. Winkler, J. B. Heymann, D. M. Belnap, W. Baumeister, and A. C. Steven. Three-dimensional structure of herpes simplex virus from cryo-electron tomography. *Science*, 302:1396–1398, 2003.

[14] H. Shan, J.P. Singh, L. Oliker and R. Biswas. Message passing and shared address space parallelism on an SMP cluster. *Parallel Computing*, 29:167–186, 2003.

[15] J. Weickert. Coherence-enhancing diffusion of colour images. *Image and Vision Computing*, 17:201–212, 1999.

[16] O. Medalia, I. Weber, A. S. Frangakis, D. Nicastro, G. Gerisch, and W. Baumeister. Macromolecular architecture in eukaryotic cells visualized by cryoelectron tomography. *Science*, 298:1209–1213, 2002.

[17] P. Perona and J. Malik. Scale space and edge detection using anisotropic diffusion. *IEEE Trans. Patt. Anal. Mach. Intel.*, 12:629–639, 1990.

[18] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 1992.

[19] M. J. Quinn. Parallel Programming in C with MPI and OpenMP. *McGraw-Hill*, 2004.

[20] J. Weickert. *Anisotropic Diffusion in Image Processing*. Teubner, 1998.

[21] J. Weickert. Coherence-enhancing diffusion filtering. *Int. J. Computer Vision*, 31:111–127, 1999.

[22] J. Weickert and H. Scharr. A scheme for coherence-enhancing diffusion filtering with optimized rotation invariance. *J. Visual Comm. Imag. Repres.*, 13:103–118, 2002.

[23] J. Weickert, B. M. ter Haar Romeny, and M. A. Viergever. Efficient and reliable schemes for nonlinear diffusion filtering. *IEEE Trans. Image Processing*, 7:398–410, 1998.

Services Integration and Task-scheduling in Bioinformatics Grids

Sergio Ramírez[a], Enrique de-Andrés[b], Ismael Navas-Delgado[c], Antonio J. Pérez[a], José Aldana[c], Oswaldo Trelles[a]

[a]Architecture of Computer Department, University of Malaga, Spain

[b]Scientist Park of Madrid, Madrid, Spain

[c]Computacional Languages Department, University of Malaga, Spain

Abstract

We report the design and structured development of a computational, data and services Grid environment at the National Institute for Bioinformatics (INB) in Spain. The design allows the easy integration of Web-based services particularly suitable for collaborative work in which users share tools, data and services. The system offers a view of public and proprietary databases as a single data source where services are readily available for enhancing data processing. The availability of a broad collection of services together with several instances of the same service on different sites allows parallel computing to be drawn over the Grid to take advantage of the huge computational resources available at the different nodes that conforms the INB. The computational load is distributed among the servers and results are coherently collected. A robust Grid-management module supplies queuing mechanisms; resources allocation, load monitoring, service-quality-based scheduling and implements fault tolerance procedures. By combining task-grained scheduling and buffering strategies to reduce idle time for load reposition and to take advantage of I/O overlapping, we have seen important increase of efficiency in the use of resources. Several tests have been performed using applications from the bioinformatics domain for which the adaptive scheduling strategy has shown its ability to produce a noticeable reduction on execution time.

Keywords: Data-service, data-Grid, bioinformatics, integration, task-scheduling, workflows

Prototype Availability: The INB prototype is available at http://www.inab.org/MOWServ/ Supplementary data on the developed tests is available at the help system.

1. Introduction

Currently, Grid computing [1] most probably represents the new generation of web-based technology. The use of such technology has promoted the development of distributed frameworks for high performance computing by means of intelligent integration of disperse and heterogeneous computational resources throughout high speed networks.

This new computing capability becomes of special interest in the bioinformatics domain where: (a) Highly diverse and impressive volume data collections are geographically disperse and have heterogeneous formats, making this plethora of interrelated information difficult to use; (b) data and services are frequently replicated in several sites; and (c) the park of installed hardware in traditional bio-wet-labs consist mainly of collections of commodity PCs. Under this scenery, Grid technology arises as a natural alternative to provide computational power to the bioinformatic community.

Several proposals have addressed the integration of bioinformatics resources. In fact, very popular sites like NCBI, EBI, ExPASy, etc.; represent a colossal effort to provide ”linked” access to a high number of interrelated data sources. However, since each service provider maintains its own web-interface style, parameter specifications and since is none standarized description about what the

input/outputs from a service are, it is difficult built up automatic service discovering and processing for the massive exploitation of Grid computational power.

Less known in the application domain but also addressing integration issues we can describe TAMBIS [15]. It makes use of an ontology that allows it to provide homogeneous layers that envelop data-bases to manage heterogeneity between sources and to provide a query interface to create and refine queries. The key idea of BioDataServer [16] for molecular database integration is a mediation architecture in which a wrapper exports some information about its source schema, data and query processing capabilities for each data source.

To meet the challenges of integrating and analysing diverse scientific data from the variety of domains within life sciences, IBM has developed a versatile platform solution, IBM DiscoveryLink [17]. With single query data access, the IBM DiscoveryLink software allows researchers to work with distributed data sources and diverse data formats. PISE [18] is a Web interface generator for molecular biology command-line driven programs, including: phylogeny, gene prediction, alignment, RNA, DNA and protein analysis, motif discovery, structure analysis and database searching programs. Its aim is to provide users with the equivalent of a basic Unix environment, with program combination, customisation and basic scripting through macro registration. EMBOSS [19] (”The European Molecular Biology Open Software Suite”) is a software analysis package specially developed for the needs of the molecular biology (e.g. EMBnet) user community. The software automatically copes with data in a variety of formats and even allows transparent retrieval of sequence data from the web.

More focused in the application domain, BioMOBY [2] is a project that proposes an architecture for the discovery and distribution of biological data, using web services. In this architecture data and services are decentralized. However, the resources are registered in a central location called MOBY central. BioMOBY objects are lightweight XML, and make up both the query and the response of a simple object access protocol (SOAP) transaction. The primary components of this architecture are MOBY Services (bioinformatics tools), MOBY Objects (input and output data in the services), MOBY Central (registry of all resources), and Object and Service hierarchies. This proposal includes the possibility of easily developing web services for publishing biological data. It introduces the use of web services for publishing and using biological data, but it is not exactly integration architecture. Taverna and Talisman [12] are in fact clients for running BioMOBY services. Both make extensive use of XML files in the Scufl representation language. Finally, myGrid [3] is an ongoing project to develop an integrated platform for bioinformatics services, with a good definition of biological objects.

On the management of Grid resources and more focused on the scheduling problem in Grid environments Nimrod-G [4] allows the distribution of multiple runs of the same process with different parameters, under static Grid configuration. This problem is solved by GRaDS scheduler [5] [6] by dynamically adapting the strategy when new resources are available. Finally, Condor-G modifies its standard scheduler to incorporate new machines connected by Globus. None of these schedulers incorporates fault tolerance concerns, neither dynamic load balancing as a function of the real capacity of each component of the Grid, these concerns are both central aspects of this work. Recently our group develop a Globus-Condor-based Grid environment in which an initial proposal were addressed [7]. Now we report a robust and stable implementation of these ideas at the INB-Spain.

We have been working from the point of view of implementing a versatile client with single and uniform user interfaces to access and present information from multiple online services and databases; and working in an internal infrastructure to integrate different and geographically distributed Web Services providing a way for fast and intuitive ”wiring” of services to create virtual

complex, distributed and powerful bioinformatics machines. Since the platform has been endowed with the ability to manage huge computational resources, dynamic scheduling and fault tolerance techniques have been used to allow massive exploitation of resources.

2. The INB web-services platform

The National Institute for Bioinformatics (INB) in Spain have addressed the integration problem throughout the design of a simple, dynamic and extensible platform to represent, recover, process, integrate and discovering knowledge. To integrate geographically distribute resources a Grid-enable system has been built on top of BioMOBY API, offering a view of the system databases as a single data source where services are readily available for enhancing data processing. Description of input/output objects is coordinated and standardised by means of an object-ontology in such a way that services can communicate among them, wiring natural bioinformatics workflows. Automatic interfaces and help system builders have been incorporated into the architecture to uniform and facilitate user communication. Beyond traditional bioinformatics platforms, data persistence system, user management and scheduling abilities have drawn a new generation of bioinformatics platforms.

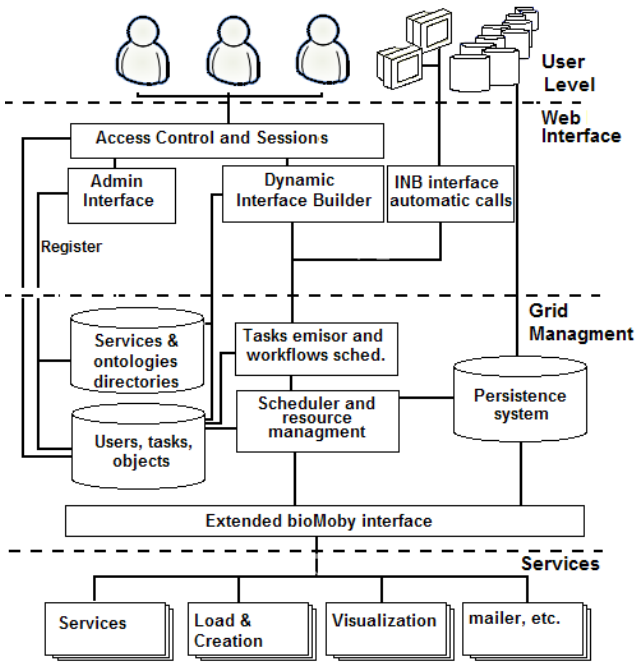


Figure 1: INB Architecture. On top, external interface including authentication and proprietary data control. Scheduling, with analysis of task-dependencies for workflows; data persistence and the capacity to use collections is offered in the Grid management level. Finally an extended BioMoby interface; including asynchronous services calls and errors management

In short the INB project aims:

- To complement and extend the BioMOBY standard to provide access to information resources (data, service and computational power), independently of a user's knowledge of their original existence.
- To combine retrieved information from these resources based on common ontologies, wiring resources and tools that in other way could not easily be linked. The workflows approach -linking several services to solve specific biological problems- opens new perspectives to the biological information analysis in Bioinformatics.
- To make interfaces simple to use, self contained and intuitive, eliminating the need for a high level of tacit knowledge. This includes displaying processed information in a consistent faceted way.
- To facilitate the incorporation of high quality, highly demanded new services we provide an internal mechanism for the efficient use of computational resources. Scalability in the platform is provided by an intelligent computational load distribution mechanism.

The scheme in 1 depicts the INB system architecture organised in three main levels, with the user minimally armed with a Web-browser, and demanding services to process their collection of biological data: (a) a web-interface on top of the architecture facilitates communication between the user and the platform (b) the architecture core including services' interface though BioMOBY API; and (c) on bottom of the scheme the services' providers.

A web interface manages user sessions with authentication mechanism. An automatic web interface builder is able to dynamically built-on interfaces for browsing data, services and namespaces (mostly associated with data containers). The catalogue offer is deployed in the form of a browseable tree from which the user gains access to procedures. In the same way automatic interfaces are built for services parameters and a generic creation service allows new objects be incorporated into the system. Up- and download procedures have also been incorporated in the platform. Noteworthy to observe, the system produce an auto-generated service help that includes training examples for getting started with each service (made available during the service registration procedure).

3. Grid Management at the INB platform

At internal level, once a service has been launched, the system provides notification about the progress status of services, -including historical record of executed tasks-, together with the relationships between input data, applied service and output data. Error notification has been incorporated in the system by extending the BioMOBY protocol. Frequently output data become the input for new services. The GUI provides a specific list of suitable services that can be applied.

Different services and multiple instances of the same service can be installed at the same or in different sites, offering computational power at different scales (ranging from simple PC servers to high cost multiprocessors platforms). Since a high number of users are expected (optimistically we look at some notable sites with more than million access per year), a pool of tasks to be solved at each moment will be the natural working scenery. Under this situation, scheduling arise as a natural need.

The Task-scheduler works over this pool of tasks and uses the map of services to choose the best server to solve the task. A task-dispatcher is in charge of launching a worker-process to communicate with the service to solve the task. Noteworthy to observe the fact that new servers and services becomes available as soon as the registry procedure is complete. Since servers can conform a sub-Grid, the scheduler transfers the job to its front-end and maintains the record of the pending jobs and the machines in charge of it. A buffering technique has been implemented, to send work in advance, avoiding delay for job reposition from the scheduler and benefit from I/O overlapping in the same machine with replicate services, which is in fact important due to bioinformatics applications are typically I/O bounded.

Load distribution is performed in a dynamic and adaptive fashion, as fast as new tasks are available. The current configuration is used to know the computational power available at a given time. The system evaluates the CPU cost of each task and adjusts predictions when the tasks are reported from the services. Load size is computed as a function of the tasks CPU cost and the quality of the service, estimated as the historic response time.

With some more detail, to perform the load distribution the following considerations are contemplated:

1. The architecture is composed of several nodes, which at the same time can form an internal sub-Grid.

2. The model is able to split a complex task in their simple components (workflows and collection of tasks).
3. Tasks are queued in a pool from which the scheduler has access to them. Tasks arrive to the queue from a tasks dependencies analyser
4. The scheduler maintains a dynamic control of the Grid configuration.
5. Fault tolerance mechanisms allow the detection of services which do not respond to the assigned task.
6. Dynamic task distribution is performed based on the quality level of the service (response time).
7. Idle time for load reposition is avoided by sending new tasks in advance (buffering).

4. Task scheduling

Work requests are analysed by a workflows and complex-task module producing individual tasks and control data dependencies among them. The scheduler works over this pool and uses a map to choose the best server to solve the task. The configuration-module dynamically traces nodes, incorporating new resources and works in close collaboration with the fault tolerance module who is in charge of the pending tasks, and it is able to re-insert tasks on the pool. Since servers can conform a sub-Grid the scheduler transfers the job to its front-end. A buffering technique [8] [9] has been implemented, to send work in advance, avoiding delay for job reposition from the scheduler. Load distribution is performed in dynamic and adaptive fashion, as fast as new tasks are available. The current configuration is used to know the computational power available at a given time. The system evaluates the CPU cost of each task and adjusts predictions when the tasks are reported from the services. Load size is computed as a function of the tasks CPU cost and the quality of the service, estimated as the historic response time (a configurable parameter).

The load distribution is estimated using the average speed observed in each service (v_i) relative to the global average speed in the system (v_{global}). The scheduler determines the size tasks (Q) to be sent to the server based on the initial task size (Q_{inic}) $Q = \frac{v_i}{v_{global}} Q_{inic}$. This distribution scheme increases the load to those servers with response-time faster than the global speed and reduces the load in the converse situation. To compute the average speed of execution the recent history is given more weight than the old one. This average speed (v_i) is computed as: $v_i = v_i' d + v_{exec}(1-d)$; where d (decay) is a configurable value (0.8 by default) related to how fast the older values are forgotten by the system, and thus it controls the velocity of adaptation of the scheduler to the changes in the quality of the service.

5. Results

The current INB interface deploys a set of services, which make possible to carry out different analysis on biological data. These different tools can be automatically applied over a set of data to produce a complete analysis to solve complex biological problems using the same platform. In this line we present a practical example to solve a phylogenetic study using an amino acid sequence as the starting point (see 2). Homology search is conducted (Blast services) to obtain similar sequences with a common evolutionary history. Output from this service contains a set of putative homologous sequences to the query. A new service is linked (Clustalw from Blast) to build-up a multiple alignment with the most similar reported sequences. Finally a phylogenetic tree is obtained using CreateTreeFromClustalw service highlighting the relations between all the sequences. Further, changing the service parameters for deeper study can customise this generic workflow.

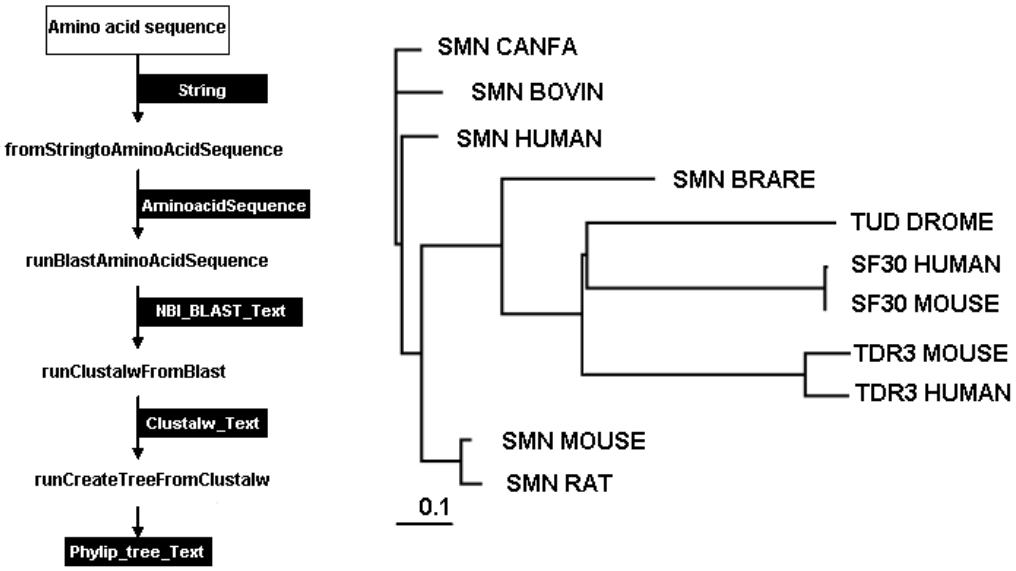


Figure 2. Homology search and phylogenetic study workflow. On the left, the grey box shows the input data; ellipses correspond to INB services, and black boxes show the input and output objects of each service in this workflow. Final and partial results remain available at the data persistence system. On the right the resulting phylogenetic tree showing the relationships among the protein sequence related to SMN_HUMAN.

This workflow has been tested with the human survival motor neuron protein (SMN; Accession Number: Q16637) running against SWISS-PROT database and a relaxed e-value was used as threshold to select distantly related homologous hits to carry out the multiple alignment. Interesting results are reported such as the fact that several related sequences show a common domain called 'Tudor domain', first identified as fragment repeats in Drosophila melanogaster [13] [14]. Surprisingly, as result of our analysis this large Drosophila protein (TUD_DROME) appears separated from the remaining proteins in the phylogenetic tree (2) suggesting a relationship with splicing factors (SF30 proteins). In short, these results can conclude that the Drosophila protein, which is required during ogenesis for the formation of primordial germ cells and for normal abdominal segmentation, could be a splicing factor assisting this process.

In the second line of work we present a parallel solution for massive Blast searches using the Grid resources at the INB. Initial tests have been performed using the computational resources available at the Scientific Park of Madrid (PCM). The PCM deploys a super cluster SGI Altix 3700 that is based on a shared memory multiprocessor architecture with 24 processors Intel Itanium2 at 1600MHz and 6Mb cache each one. The memory and disk storage are 40Gb and 1Tb respectively. The PCM also has three Dual Intel Xeon Servers at 2.8 GHz with 1 GB of RAM memory.

In this test we have used the sequence of the bacterium *Buchnera aphidicola str. Bp* (Baizongia pistaciae) with Accession Number NC_004545, whose proteome has 504 proteins and it is considered one of the known organisms with the minimal gene number needs for symbiotic life. In this way this proteome constitutes a diverse set of proteins representing roughly the different amino acid sequences in the nature.

Two experiments have been conducted to study the (a) effect of using a parallel Blast implementation to solve each task; (b) the influence of the task and problem size. In both cases we have used advanced delivery of jobs (to avoid idle time) and the fault-tolerance module was active (to complete

the set of tasks and make the experiments comparable). The main results can be drawn as follow: (1) execution time is significantly reduced even for single sequences tasks, until a number of processors are enough (6 for a medium size sequence of 400 aa). However, the combined use of parallel Blast and parallel tasks allows a high number of PEs to be used with promising results in reduction time.

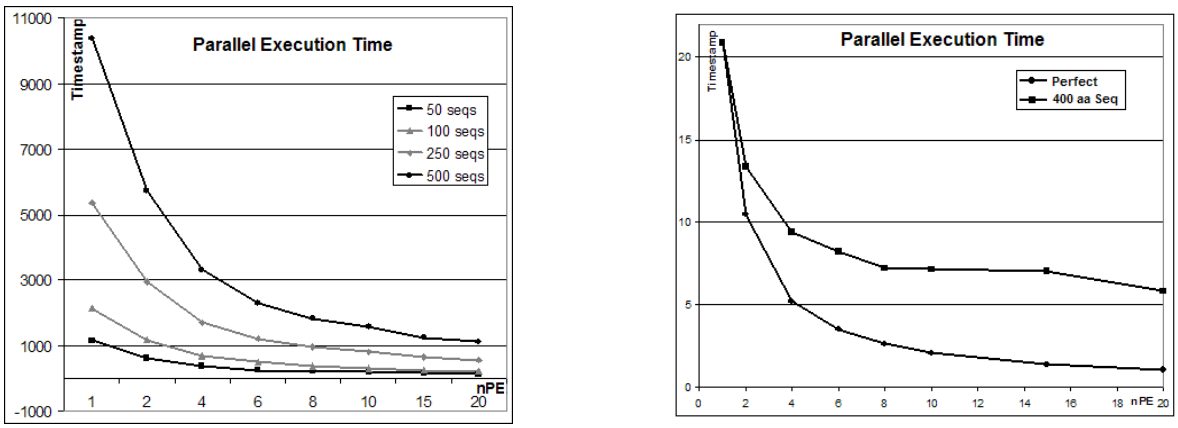


Figure 3. Parallel execution time, using parallel Blast implementation (on the left) and combining parallel Blast and parallel tasks in different number of processors (on the right).

As expected, for low load (50 and 100 sequences datasets) the benefits of using multiprocessors are limited due to the overhead produced by communication time. Results compares quite well against those obtained using single-processors with high efficiencies.

6. Conclusions

In summary we have developed a dynamic and flexible integration architecture that provide fast and intuitive ”wiring” of services thus creating virtual complex, distributed and powerful bioinformatics machines. Based on semantic interconnection concepts the platform integrates into workflows diverse traditional databases and various processing mechanisms developed by different users and groups through a web-based interface expanding the functionality of current services and enabling the easy incorporation of new procedures to customize the system for specific concerns. Additionally by supplying computational power the INB architecture makes easier the challenge of discovering new knowledge from the avalanche of new biological data.

An adaptive scheduling for task-grained execution in Grid environments has been presented. The scheduling policy has been applied to the resolution of a common bioinformatics problem and each scheduling module has been individually analysed for their influence on efficiency improvement: task buffer size, task size; initial load size; fault-tolerance, etc. These results compare quite well with those obtained using static scheduling with very encouraging results.

The usefulness of the INB-architecture is demonstrated both at the level of integrating diverse services to produce a complete view of a given biological process and to obtain results with a significant reduction of elapsed time compared with equivalent installations.

7. Acknowledgements

This work has been partially supported by grant ”GNV5-Bioinformática Integrada” from Genoma-España.

References

[1] I. Foster, C. Kesselman, and S. Tuecke: The anatomy of the grid: Enabling scalable virtual organizations. Supercomputer Applications, 2001 (http://www.globus.org/research/papers/anatomy.pdf).

[2] Wilkinson, MD, Links, M.: ”BioMOBY: an open-source biological web services proposal”.Briefings In Bioinformatics 3:4. 331-341.2002 (www.biomoby.org)

[3] R. Stevens, A. Robinson y C. A. Goble: ”myGrid: Personalised Bioinformatics on the Information Grid”. In proceedings of 11th International Conference on Intelligent Systems in Molecular Biology. Brisbane, Australia. Bioinformatics vol. 19 Sup. 1 2003.

[4] Abramson, D.: ”High performance parametric modelling with Nimrod/G: Killer application for the global Grid?”. In proceedings of the Seventh International Symposium on High Performance Distributed Processing Symposium (IPDPS’00).2003

[5] Berman, F.: ”The GrADS Project: Software support for high-level Grid application development”. International Journal of Supercomputer Applications 15, 4, 327-344. 2001

[6] H. Dail: ”A Decoupled Scheduling Approach for the GrADS Environment”. Proceedings of SC 2002, Baltimore.2002.

[7] Manuel Hidalgo Conde, Andr’es Rodr’iguez, Sergio Ram’irez and Oswaldo Trelles: ”Adaptive Task Scheduling in Computational GRID environments”. European GRID Conference; Amsterdan, Holland.2005

[8] Trelles-Salazar, O.; Zapata. E. and Carazo, J.M.: ”Mapping Strategies for Sequential Sequence Comparison Algorithms on LAN-based Message Passing Architectures”. High Performace Computing and Networking (HPCN-Europe’94), Munich-Alemania.1994

[9] Trelles-Salazar, O. Zapata, E.L. and Carazo J.M.: ”On an efficient parallelization of exhaustive sequence comparison algorithms”. Computer Applications in BioSciences 10(5):509-511.1994

[10] Rodriguez, A. Prez-Pulido, A:, Thode,G.; Carazo,JM. y Trelles,O.: ”Mining Low-level similarity signals from sequence databases”.High Performance Computers Applied to BioInformatics and Computational Biology. 4th Conference on Systematics, cybernetics and Informatics, SCI’2000; Orlando, Florida, USA.2000

[11] Altschul SF, Madden TL, Schaffer AA, Zhang J, Zhang Z, Miller W, Lipman DJ.: ”Gapped BLAST and PSI-BLAST: a new”.1997

[12] Oinn, Tom; Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R. Pocock, Anil Wipat and Peter Li: ”Taverna: A tool for the composition and enactment of bioinformatics workflows Bioinformatics Journali”.20(17) pp 3045-3054.2004

[13] Ponting C.P.: ”Tudor domains in proteins that interact with RNA. Trends Biochem”. Sci. 22:51-52. 1997

[14] Callebaut I., Mornon J.P.: ”The human EBNA-2 coactivator p100: multidomain organization and relationship to the staphylococcal nuclease fold and to the tudor protein involved in Drosophila melanogaster development”. Biochem. J. 321:125-132. 1997

[15] Stevens et alt.: ”TAMBIS: Transparent Access to Multiple Bioinformatics Information Sources”. Bioinformatics, 16:2 PP. 184-186.

[16] Lange et alt.: ”A computational Support for Access to Integrated Molecular Biology Data”.

[17] DeCarlo, J.: ”IBM Life Sciences Solutions: Turning Data into Discovery with DiscoveryLink”. ISBN 0738423254. 2002

[18] Letondal, C.: ”A Web interface generator for molecular biology programs in Unix. Bioinformatics”, Oxford University Press, 17(1), pp 73-82. 2001

[19] Rice,P. et al.: ”EMBOSS: the european molecular biology open software suite”. Trends Genet., 16, 276-277. 2000

Minisymposium

Network-On-Chip

Networks on Chips: A Synthesis Perspective

Federico Angiolini[a], Paolo Meloni[b], Davide Bertozzi[c], Luca Benini[a], Salvatore Carta[d], Luigi Raffo[b]

[a]Dipartimento di Elettronica, Informatica e Sistemistica, University of Bologna, 40136 Bologna, Italy

[b]Dipartimento di Ingegneria Elettrica ed Elettronica, University of Cagliari, 09123 Cagliari, Italy

[c]Dipartimento di Ingegneria, University of Ferrara, 44100 Ferrara, Italy

[d]Dipartimento di Matematica e Informatica, University of Cagliari, 09123 Cagliari, Italy

To face increasing requirements for computational density in embedded chips, MultiProcessor Systems-on-Chip (MPSoCs) are being widely deployed. This evolution increases communication requirements, therefore new, more scalable, on-chip interconnect fabrics are being called for. Networks-on-Chip (NoCs) appear to solve the upcoming scalability issue. However, it is presently unclear how exactly NoCs can position themselves in terms of performance/area tradeoff.

Since NoCs are supposed to span across the whole chip area, difficult questions associated to chip layout arise. Specifically, the delay impact of long-range wiring resources is unknown, and the delay estimation provided by synthesis tools must be verified against post-placement figures.

This paper will address such question marks, by showing a complete NoC synthesis flow going down to the layout level. The experimental results include assessment of a complete placed NoC instance and analysis of single switches when synthesized in varying configurations.

1. Introduction

The ever increasing System-on-Chip (SoC) integration is calling for a huge increase in intra-chip communication resources. The traditional SoC interconnect fabric leverages upon a single shared bus; this approach is quickly becoming insufficient to cope with current and future requirements. More complex and better performing architectures, namely crossbars, are being studied and deployed, but they are inherently non-scalable due to wiring congestion. This fact dictates the need for a more radical departure from the common notion of a monolithic communication fabric featuring a fixed topology. Instead, packet-switched networks, such as the Internet, spring to mind for their efficiency and inherent scalability. The transfer of the packet-switching concept to the on-die realm takes the name of Network-on-Chip (NoC). NoCs provide a flexible path toward complex fabric topologies with abundant amounts of available bandwidth.

SoC architectures exhibit a highly heterogeneous nature, and industry trends point at the integration of an increasing number of different functional blocks on the same die. Therefore, homogeneous or preconfigured NoCs look like an inefficient solution to the interconnect scalability problem. This observation calls for the creation of a library of highly parameterizable NoC components, enabling the designer to instantiate a NoC architecture custom-tailored to specific applications and usage needs. The configurability of such “soft macros” however must be provided without incurring in any noticeable performance overhead.

While undertaking the effort of developing such building blocks, it is of course crucial to assess the performance, area and power targets of the whole NoC instance. This task is not trivial, also

due to the unknown wiring overhead after performing the final chip placement and routing. Such overhead is clearly critical in a distributed component spanning the whole chip area. Thus, a key design principle must be a strict control of wiring resources, so that they do not span across long distances on the die. This can be achieved *e.g.* by link pipelining through clocked repeaters. In any case, it is essential to be able to carry the design through all stages of physical synthesis before being able to properly estimate critical factors such as maximum achievable clock frequency. The presently available industrial and academic literature is lacking thorough examples of this analysis.

This work will discuss the above key topic by presenting synthesis results for ×pipes, a library of fully synthesizable, highly customizable, high frequency and low latency NoC modules. By leveraging upon a flexible custom-built CAD toolchain, we enabled a complete flow where the ×pipes components were deployed in a NoC topology interconnecting functional IP cores. This topology, which is fully simulatable to assess architectural efficiency, was then synthesized in a 0.13 μm technology, and finally placed and routed. The last step allowed us to get reliable figures about critical parameters such as frequency and area of the NoC. In this work, we will mostly focus on the description of a complete topology instantiation and on the area/delay issues of wiring resources. A thorough power consumption analysis, while possible in the framework that we will describe, requires additional research to identify appropriate traffic patterns for the modeling of an entire topology and of single building blocks, and is therefore left for future work.

This paper is organized as follows. Section 2 will discuss the state of the art in NoC implementations. Section 3 will describe the component blocks of the NoC we developed and synthesized. Section 4 will describe the synthesis steps and related issues, while Section 5 will show experimental results. Finally, Section 6 will draw conclusions.

2. Related Work

To face the interconnect scalability bottleneck, NoCs have been suggested as an effective long-term solution [8], [3]. In recent years, an increasingly large body of research has bloomed around this subject, focusing for example on complete architectures [11], flow control protocols [13], Quality of Service (QoS) provisions [6], [16], support CAD tools [4], asynchronous implementations [5]. While these issues are critical, they are often discussed without a complete analysis of the area and delay implications when carrying such designs onto a real chip. Work focused on the synthesis stage of the NoC flow exists. For example, a test chip, mostly focused on electrical properties and power consumption, is mentioned in [12], while fully placed designs are presented in [14], [2]. However, neither of them proposes a completely configurable library of parameterizable building blocks, neither presents wide spectrum architectural exploration results, and all rely on some full custom blocks to achieve higher performance. For some NoCs [17], FPGA mappings are described in the literature. However, these attempts help shedding light only up to a certain extent, due to the huge unpredictability of the FPGA synthesis results across different device families and vendors. Moreover, physical delay issues get masked by the architecture of the specific FPGA chip at hand.

We build on our previous work [7], [15] to further illustrate a fully parametric NoC library, carrying its implementation to the layout level. While still being able to explore architectural design tradeoffs thanks to the completely synthetic nature of the flow, this step makes it possible to evaluate important design assumptions, such as wire predictability and scalability.

3. ×pipes Architectural Blocks

×pipes is a highly configurable, high performance NoC. One of its most prominent features is the ability to be deployed in different topologies and with different design parameters, thus matching the requirements of any specific target application. To achieve this objective, ×pipes is based upon a library of components; it is the system designer's responsibility to pick up, configure and connect the proper blocks to create the complete interconnect fabric, of course assisted by CAD tools. The ×pipes library contains three main components: switches, Network Interfaces (NIs) and links.

Switches constitute the backbone of the NoC, as they route packets from sources to destinations. They can be connected in a flexible manner, according to the designer's preference, thus resulting in arbitrary fabric topologies. Switches provide buffering resources to improve performance; in ×pipes, output buffering was chosen, *i.e.* FIFOs are present on each output port. Moreover, switches handle flow control issues, and resolve conflicts among packets when they overlap in requesting access to the same physical links.

NIs take care of the task of converting processor transaction requests into packets, and of the dual process of transaction unpacketing. Therefore, an NI is needed to connect each IP core to the NoC. Further, each packet is split by the NIs into a sequence of "flits" (FLow control unITS) before transmission, to decrease the physical wire parallelism requirements. In ×pipes, two separate NIs are defined, an *initiator* and a *target* one, respectively associated to system masters and system slaves. In case of a device requiring both interfaces, an NI of each type must be attached to it. The interface among IP cores and NIs is defined by the OCP 2.0 [1] specification, for maximum reuse capability. NIs also take care of specifying the path that packets will follow in the network to reach their destination (source routing), by means of path Look-Up Tables (LUTs). NIs provide dual clock capability, in that two different clock signals can be attached to it: one is driving the NI front-end (the OCP interface), the other is driving the NI back-end (the ×pipes interface). These clocks have to be in an integer multiple relationship with one another, the OCP clock being slower. This arrangement allows the NoC fabric to run at a fast clock even though some or all of the attached IP cores are slower, which is crucial to keep transaction latency low. Also, each IP core can run at a different divider of the ×pipes frequency, therefore making mixed-clock platforms possible.

Inter-block links are a critical component of NoCs, especially when taking into account technology trends. Since links can span across the whole die or significant portions thereof, the problem of signal propagation delay is critical. For this reason, ×pipes supports link pipelining, *i.e.* the interleaving of logical buffers along links. Proper flow control protocols are implemented in link transmitters and receivers, *i.e.* within NIs and switches, to make the link latency transparent to the surrounding logic. By means of this design choice, the overall platform can run at a fast clock frequency, without the longest wires being a global speed limiter. Instead, only the links which are too long for single-cycle propagation will pay a latency penalty.

A custom CAD tool, called ×pipesCompiler, allows the designer to instantiate and configure ×pipes library components to fit his or her application needs. By means of a simple specification text file (which may be generated as the output of an additional external tool), components can be attached to each other in arbitrary topologies, and routing tables can be freely set. The NoC flow control policy can be chosen among a simple single-token credit-based one (called STALL/GO), and a retransmission-enabled one (called ACK/NACK), which provides some of the facilities required for more thorough fault tolerance. The flit width can be set to arbitrary values. Additionally, switches can be customized in terms of I/O ports, buffering resources and arbitration policies, while network interfaces in terms of buffering resources and clock dividers.

4. Toward Chip Layout

One of the aims of the ×pipes NoC infrastructure is devising a complete design flow, letting the designer configure, simulate, synthesize and verify an MPSoC instance. For this paper, we will especially focus on the portions of this flow which pertain to the synthesis and place&route stages.

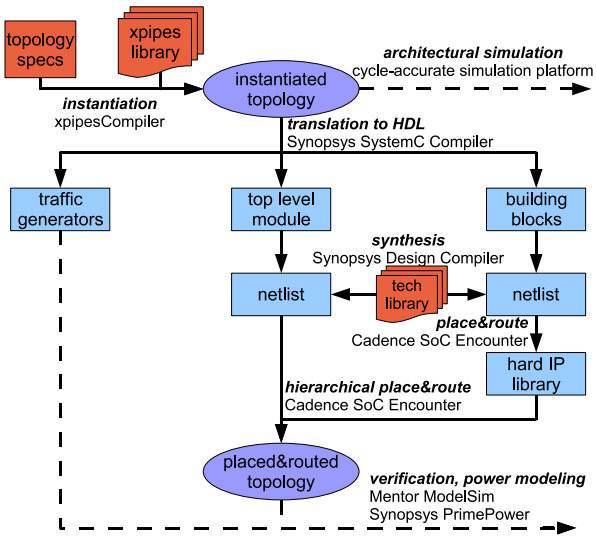
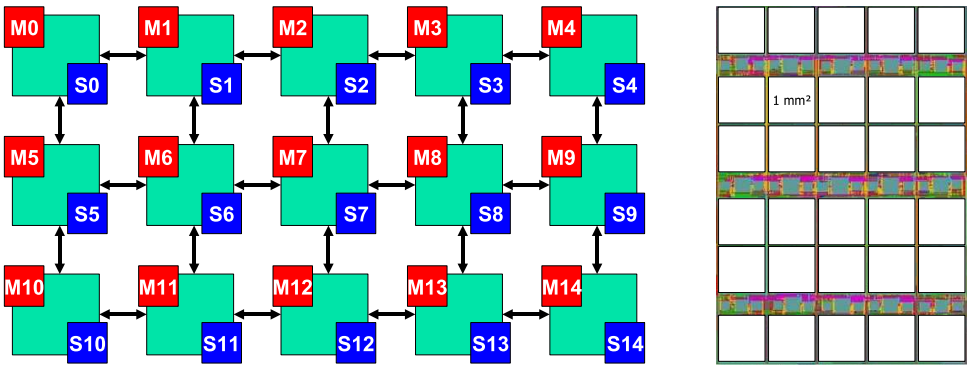


Figure 1. The ×pipes design flow

Figure 1 is showing a high-level view of the complete process of instantiating a NoC platform. The main inputs of the flow are the ×pipes component library, written in SystemC, and a text format topology specification. The ×pipesCompiler tool processes these sources to generate a SystemC instance, which is suitable both for architectural simulation and for synthesis. When the synthesis path is taken, a SystemC-to-HDL automatic translation is performed. Three major outputs are the result: the HDL code for traffic generators, for the NoC building blocks, and for the top level instantiation code. The latter two are synthesized with Synopsys Design Compiler [10]. During this stage, a technology library is required; for this paper, we used a 0.13 μm library. No full custom blocks are employed in the process, even though they would lead to better performance, to illustrate the results achievable with a completely synthetic, and therefore maximally flexible, approach. The resulting netlists now need to be placed and routed, which is done in two steps by Cadence SoC Encounter [9]. First, each building block is processed; subsequently, blocks are connected together according to the top level module specifications, and the final optimizations are made. The final output is a placed&routed topology, which is suitable for further area and delay analysis and for verification. By back annotation of parasitic capacitances, power figures could be extracted, but such analysis is beyond the scope of this paper.

5. Experimental Results

Figure 2(b) depicts a sample NoC floorplan resulting from the previously described flow. For this paper, we chose a 5x3 regular mesh (Figure 2(a)), which interconnects 30 IP cores (15 masters and



(a) (b)

Figure 2. Topology (a) and floorplan (b) of a 5x3 ×pipes mesh

15 slaves). Each of these cores is supposed to be square and obstructing an area of 1 mm^2, which is reasonable for embedded cores and/or 32 kB SRAM banks at the 0.13 μm node. Synthesis was carried on by leveraging a UMC 0.13 μm technology library, which does not have extreme performance as its focus, but for which obstruction information was available to us in the place&route phase. Our test topology comprises 30 NIs, equally split among initiators and targets, and 15 switches, of which three having 6 input and 6 output ports (those in the center of the mesh), eight 5x5 (at the sides), and four 4x4 (in the corners). Each switch is connected to one initiator and one target NI. The mesh is configured with 38-bit flits, 3-flit buffering FIFOs, fixed priority arbitration, the ACK/NACK flow control protocol, and non-pipelined links. Experimental results for this topology are reported in Figure 3 at varying frequency targets. As can be seen, even when pushed to the performance limits, the NoC cells do not require more than 7% of the overall die area. At a lower frequency target of 500 MHz, NoC cells take 5.7% of the chip real estate. However, wiring resources and the need to reserve some slack space for the global nets (*e.g.* the clock tree) cause the baseline area of 30 mm^2 (30 IP cores of 1 mm^2 each) to stretch to a global area of 42 to 44 mm^2. We would like to mention that we did not perform any specific area optimization in this work.

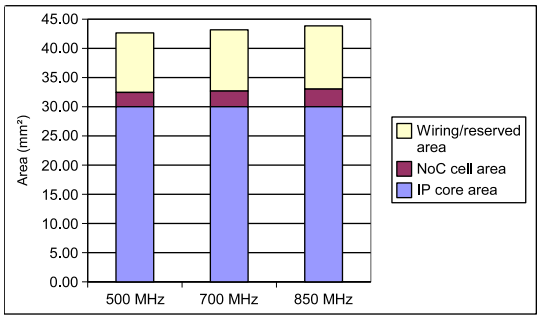


Figure 3. Area of a 5x3 ×pipes mesh synthesized with different frequency targets

An important figure to notice here is the degree of divergence among delay estimations based upon the netlist and upon the placed topology. When aiming for maximum frequency, the netlist was estimated to be capable of running at 910 MHz; after the place&route stage, the real achievable frequency was 845 MHz, which translates into a decrease of just 8%. Moreover, the critical path was within the NoC logic (namely, the 6x6 switches), and not on the global wires, which even allowed us to deploy non-pipelined links among the NoC blocks. These findings confirm the effectiveness of NoCs in downplaying global wires as performance limiters.

Figure 4 analyzes the behaviour of switch components when synthesized with varying configura-

tion parameters. We will explore the configuration space independently for each parameter, keeping as a baseline a 6x6, 38-bit switch having a 3-flit buffer depth and using fixed priority arbitration and ACK/NACK flow control. All instances have been synthesized, placed and eventually routed to provide figures as accurate as possible.

Figures 4(a) and 4(b) show trends when varying the amount of I/O ports; since the bulk of the switch logic and buffering in ×pipes is associated with output ports, which is compounded by a mild dependence on inputs, the area scales up a bit more than linearly with the amount of output ports. Increasing numbers of input ports make arbitration and multiplexing more complex, which results in a 10% worse frequency when moving from four to six inputs.

Figures 4(c) and 4(d) depict performance when varying the flit width of packets. When moving from 16 to 38 bits (which is an optimal flit size for performance once the decomposition of packets into flits is taken into account), a huge area penalty of 64% can be observed. Such penalty is however weakly proportional to flit width, which increased by 138%. This result is logical, since the area for the datapath (including buffers) has to scale linearly with flit width, but arbitration and control logic are unaffected. The maximum operating frequency is also almost unaffected by flit width, which suggests the worsening of wiring congestion to be not critical.

The impact of buffering is investigated in Figures 4(e) and 4(f). Since buffering resources represent a significant percentage of the component area, doubling the buffer depth results in a noticeable 54% area penalty. Also, due to the FIFO nature of the buffers, increased logic and wiring complexity impacts maximum frequency by as much as 52 MHz (around 6%).

Implementing a fair round-robin arbitration policy instead of the baseline fixed priority one incurs a noticeable cost. Additional logic to track the status of input ports results in 15% worse area and maximum operating frequency (Figures 4(g) and 4(h)).

The choice of flow control protocols impacts the performance of the NoC also with respect to performance of the single building blocks. We tested with two alternative protocols, namely ACK/NACK (which features retransmission capabilities, and is suitable for fault-tolerant environments) and STALL/GO (which is a variant of credit-based schemes where only one credit is available, thus allowing for a pipelined link implementation). The ×pipes architecture natively exhibits output buffering, and is therefore more suitable to the ACK/NACK protocol. When deploying STALL/GO logic, additional input buffering had to be added; albeit minimal, this incurred a 52% area overhead and a 5% lower frequency (Figures 4(i) and 4(j)).

6. Conclusions and Future Work

In this paper, we showed a complete NoC synthesis flow leading to chip layout. Synthesis results highlight the relationship among design parameters and final performance, helping the designer to choose the best tradeoffs. As an example, for the ×pipes architecture, to achieve maximum performance if area is not a concern, increasing the flit width first is better than adding buffers, since it has a much lower frequency penalty.

The place&route phase is essential to validate the proposed flow and to check design assumptions. Despite of the distributed nature of NoC fabrics, post-place&route synthesis figures do not show a significant delay degradation with respect to netlist estimates, and critical paths are within the building blocks. When considering that traditional interconnects are limited by global wire delay already at the current lithographic nodes, these two findings strengthen the positioning of NoCs as a highly scalable, highly predictable interconnect fabric for future technologies.

Future work revolves around architectural optimizations, a thorough power consumption analysis and the investigation of the performance impact of the usage of full custom logic blocks.

References

[1] Open Core Protocol Specifi cation, Release 2.0, http://www.ocpip.org, 2003.

[2] Adrijean Andriahantenaina and Alain Greiner. Micro-network for SoC : Implementation of a 32-port spin network. In *The Proceedings of Design, Automation and Test in Europe Conference and Exhibition*, pages 1128– 1129. IEEE, 2003.

[3] Luca Benini and Giovanni De Micheli. Networks on chips: A new SoC paradigm. *IEEE Computer*, 35(1):70 – 78, January 2002.

[4] Davide Bertozzi, Antoine Jalabert, Srinivasan Murali, Rutuparna R. Tamhankar, Stergios Stergiou, Luca Benini, and Giovanni De Micheli. NoC synthesis fbw for customized domain specifi c multiprocessor systems-on-chip. *IEEE Transactions on Parallel and Distributed Systems*, 16, Issue 2:113–129, February 2005.

[5] Tobias Bjerregaard and Jens Sparsø. Scheduling discipline for latency and bandwidth guarantees in asynchronous network-on-chip. In *Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 34–43, 2005.

[6] Evgeny Bolotin, Israel Cidon, Ran Ginosar, and Avinoam Kolodny. QNoC: QoS architecture and design process for network on chip. In *Journal of Systems Architecture*. Elsevier, 2004.

[7] Matteo Dall'Osso, Gianluca Biccari, Luca Giovannini, Davide Bertozzi, and Luca Benini. ×pipes: A latency insensitive parameterized Network-on-Chip architecture for multi-processor SoCs. In *Proceedings of 21st International Conference on Computer Design*, pages 536–539. IEEE Computer Society, 2003.

[8] William J. Dally and Brian Towles. Route packets, not wires: On-chip interconnection networks. In *Proceedings of the 38th Design Automation Conference*, pages 684–689, June 2001.

[9] Cadence Design Systems Inc. SoC Encounter, http://www.cadence.com.

[10] Synopsys Inc. Design Compiler, http://www.synopsys.com.

[11] Faraydon Karim, Anh Nguyen, Sujit Dey, and Ramesh Rao. On-chip communication architecture for OC-768 network processors. In *Proceedings of the Design Automation Conference (DAC)*, pages 678 – 683, 2001.

[12] Kangmin Lee, Se-Joong Lee, Sung-Eun Kim, Hye-Mi Choi, Donghyun Kim, Sunyoung Kim, Min-Wuk Lee, and Hoi-Jun Yoo. A 51mW 1.6GHz on-chip network for low-power heterogeneous SoC platform. In *Digest of Technical Papers of the 2004 IEEE International Solid-State Circuits Conference (ISSC)*, pages 152–518. IEEE Computer Society, 2004.

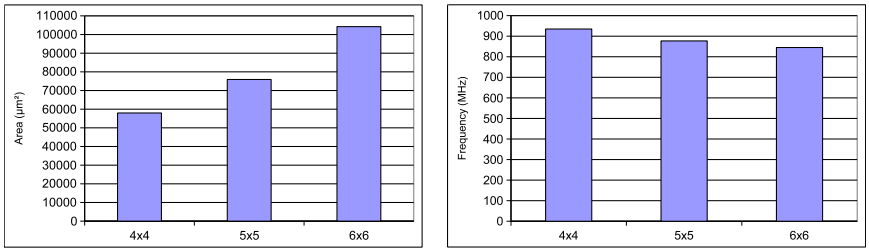
[13] Antonio Pullini, Federico Angiolini, Davide Bertozzi, and Luca Benini. Fault tolerance overhead in network-on-chip fbw control schemes. In *Proceedings of the SBCCI Conference 2005 (to be published)*, 2005.

[14] Andrei Radulescu, John Dielissen, Kees Goossens, Edwin Rijpkema, and Paul Wielage. An effi cient on-chip network interface offering guaranteed services, shared-memory abstraction, and fexible network confi guration. In *Proceedings of the 2004 Design, Automation and Test in Europe Conference (DATE'04)*. IEEE, 2004.

[15] Stergios Stergiou, Federico Angiolini, Salvatore Carta, Luigi Raffo, Davide Bertozzi, and Giovanni De Micheli. ×pipes Lite: A synthesis oriented design library for networks on chips. In *Proceedings of Design, Automation and Testing in Europe Conference 2005 (DATE05)*, pages 1188–1193. IEEE, March 2005.

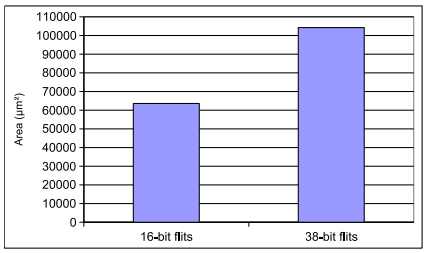
[16] Daniel Wiklund and Dake Liu. SoCBUS: Switched network on chip for hard real time embedded systems. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS03)*. IEEE, 2003.

[17] Cesar Albenes Zeferino and Altamiro Amadeu Susin. SoCIN: A parametric and scalable network-on-chip. In *Proceedings of the 16th Symposium on Integrated Circuits and Systems Design (SBCCI03)*, pages 34–43, 2003.

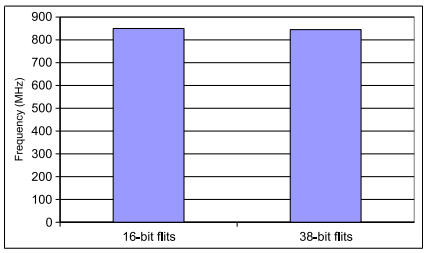


(a)

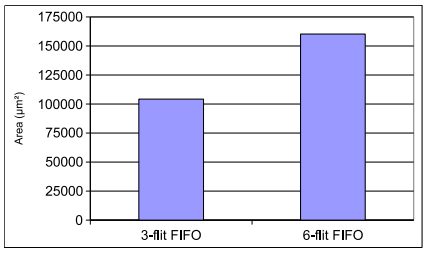
(b)



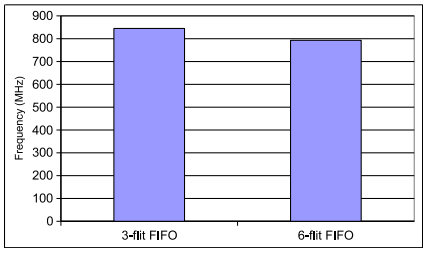
(c)



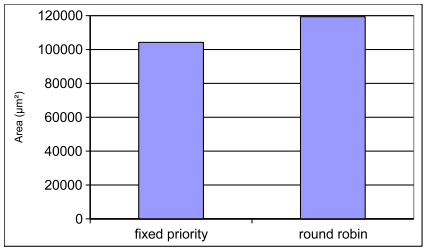
(d)



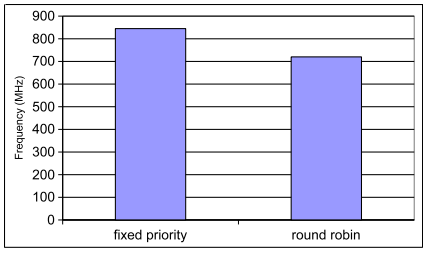
(e)



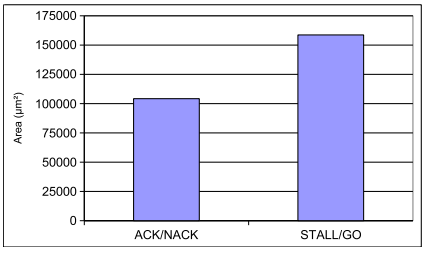
(f)



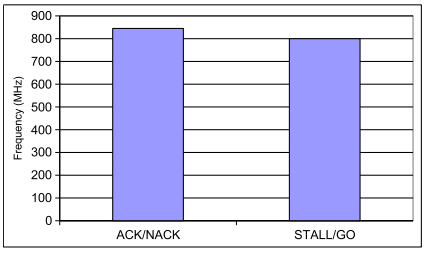
(g)



(h)



(i)



(j)

Figure 4. Area and max frequency of ×pipes switches configured with varying parameters: number of ports (a-b), flit width (c-d), buffering (e-f), arbitration policy (g-h), flow control protocol (i-j)

NoC Emulation on FPGA: HW/SW Synergy for NoC Features Exploration

Nicolas Genko*, David Atienza*†, Giovanni De Micheli*
* LSI/EPFL, EPFL-IC-ISIM-LSI Station 14, 1015 Lausanne, Switzerland.
E-mail: {nicolas.genko, david.atienza, giovanni.demicheli}@epfl.ch
†DACYA/UCM, Avda. Complutense s/n, 28040 Madrid, Spain.

Abstract. Current Systems-On-Chip (SoC) execute applications that demand extensive parallel processing. Networks-On-Chip (NoC) provide a structured way of realizing interconnections on silicon, and largely alleviate the limitations of bus-based solutions. NoCs can have regular or ad hoc topologies, and functional validation is essential to assess their correctness and performance. In this paper, we present a flexible emulation environment implemented on an FPGA that is suitable to explore, evaluate and compare a wide range of NoC solutions with a very limited effort. We also present an automated way to perform NoC features exploration using the interaction HW/SW on an FPGA. Our experimental results show a speed-up of four orders of magnitude with respect to cycle-accurate HDL simulation, while retaining cycle accuracy. With our emulation framework, designers can explore and optimize a various range of solutions, as well as quickly characterize performance figures.

1. Keywords

HW/SW Co-design, Interconnection mechanisms, embedded systems, Networks-on-Chip.

2. Introduction

With the growing complexity in consumer embedded products, new tendencies envisage heterogeneous *System-On-Chip* (SoC) architectures consisting of complex integrated components communicating with each other at very high-speed rates. Intercommunication requirements of SoCs made of hundreds of cores will not be feasible using a single shared bus or a hierarchy of buses due to their poor scalability with system size and their shared bandwidth between all the cores attached to them.

To overcome these problems of scalability and complexity, *Network-On-Chip* (NoC) has been proposed as a promising replacement for buses and dedicated interconnections [3,10]. NoCs involve the design of network interfaces to access the on-chip network and switches to provide the physical interconnection mechanisms to transport the data of the cores. Therefore, the definition and implementation of NoCs involve a complex design process, for instance, the selection of suitable protocols or topologies of switches to use.

Concrete options for NoC topologies and interfaces have been proposed at different levels of abstraction [15,11,12,4] and some even implemented onto FPGAs for functional validation. Nevertheless, these different physical implementations onto FPGAs are limited in flexibility and do not enable complete tests of different actual realizations of NoC on silicon.

In this paper, we illustrate the benefits of a complete mixed HW-SW NoC emulation platform [1,2] where a wide range of NoC features can be easily instantiated and compared at physical level. Such emulation is possible since our emulator takes a NoC without modifying the device under test. As a result, this emulation framework provides a consistent way to test the performance achieved by actual physical realizations of NoCs on silicon at a very high speed (16000 times faster than an HDL simulator). It is implemented onto an FPGA board and supplies a wide range of statistics for the different traffic patterns typically generated in NoCs. In addition, our framework implementation is very modular and the statistics reports are easily extensible for further testing of concrete effects (e.g. saturations effects in parts of NoCs) on a particular NoC instantiation.

The remainder of the paper is organized as follows. In Section 3, we describe some related work.

In Section 4, we summarize the architecture of our emulation framework and its HW/SW flows [1,2]. In Section 5, we show for the first time how to apply the concept of emulation that we have developed at different level of NoC Design. In Section 6 we present an algorithm to perform NoC design exploration with our emulation framework and provide an example of its use. Finally, in Section 7 we draw our conclusions.

3. Related Work

In the last years, significant research has been done to evaluate the design and implementation features of NoC at its different levels of abstraction. To provide accurate functional validation (i.e. circuit level), several approaches have been implemented in FPGAs. In [12] and [4], NoCs with a mesh-based topology and packet-switching as communication mechanism have shown the effectiveness of NoC. Also, other NoC architectures (e.g. torus) and designs of switches/routers have been ported to FPGAs in order to validate their NoC features (e.g. packet sizes, switching-mode) based on additional HDL simulations [13,20,21,6]. These previous approaches can validate several NoC implementations features, but none of them is designed to exhaustively test the details of NoC topologies and traffics as ours.

To evaluate in detail different architectural alternatives reducing the cost of synthesizable NoC design, several cycle-accurate simulation environments have appeared. In [17], VHDL is employed to evaluate several features of virtual channels in mesh-based and hierarchical NoC topologies. In [15], XML and SystemC are used to specify the NoC components (e.g. routers, network interfaces) and to test mesh-based NoC design alternatives. The main difference with our approach is that their simulations have a much larger execution time compared to our physical NoC emulation environment.

To increase the simulation speed of cycle-accurate VHDL, several approaches have been proposed. [9] and [11] describe modeling environments for custom NoC topologies based on SystemC. [5] presents a mixed VHDL/SystemC implementation and simulation methodology using a template router to support several interconnection networks. In [7], a C++-based library of communication APIs is built on top of SystemC to explore NoCs topologies. Finally, [14] presents a fast transaction level modeling approach to explore bus-based communication architectures. While the previous approaches enable the fast exploration of the main features of NoC designs as our proposed emulation platform, their level of accuracy in the estimations and their simulation speed is more limited compared to our complete physical emulation of parameterizable NoCs.

Other proposed approaches improve the speed of cycle-accurate NoC simulations by using high-level abstraction languages, e.g. C or C++. [8] presents a C-based interconnection network simulator. Similarly, [19] proposes an event-driven C++ simulator. Also, [10] presents a NoC design methodology that uses a parameterizable NoC architecture executed in a high-level event simulator. At a higher-level of abstraction, several algorithms, analytical models and heuristics have been proposed to achieve very fast rough estimations of the cost of NoC topologies based on graphs representions [18,16]. Although these approaches attain high simulation speeds (sometimes close to real hardware), they cannot obtain detailed statistics of final physical implementation systems as our emulation framework does.

4. Overall Emulation Framework

Our emulation approach has been designed in a modular way to easily implement various custom NoC topologies and architectures. An overview of the architecture of our framework is depicted in Figure 1. It consists of three main elements, which are mapped onto an FPGA board with a hard-core processor. In this case, we have used a Xilinx Virtex 2 Pro v20 and a Power PC. The hard-core processor of the FPGA is used to orchestrate the emulation process in a flexible way. Then, the monitor module provides the interface to communicate with the host PC and to show the produced statistics onto its screen through the serial port. Finally, the main element of our NoC emulation framework is the NoC programmable emulation platform. It is a module that consists of the necessary elements

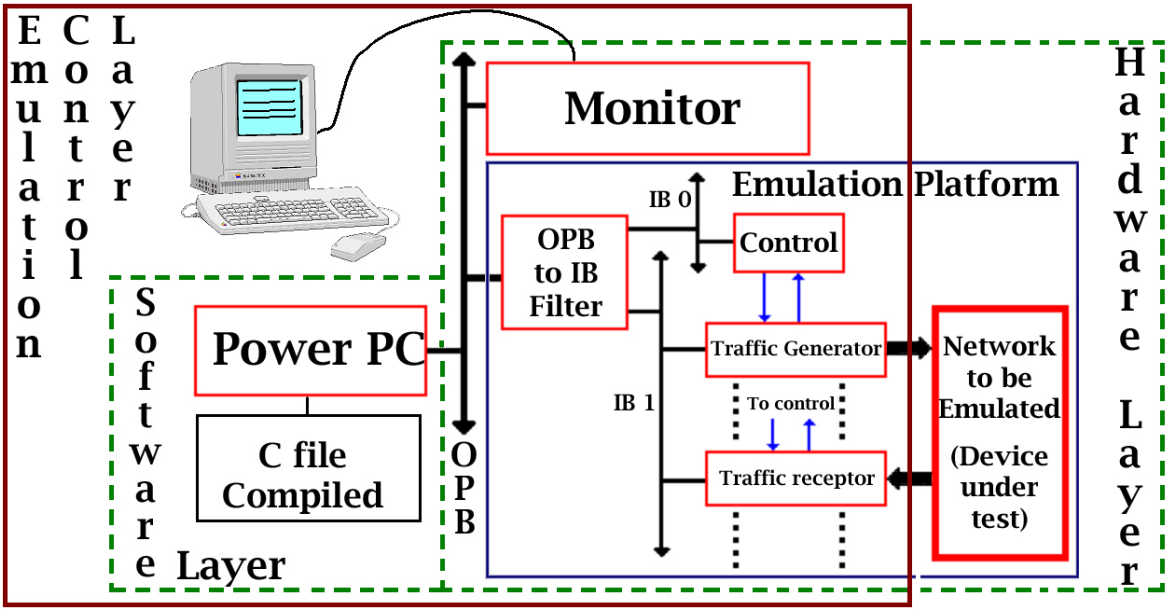


Figure 1. NoC Emulation Framework

to emulate the device under test. Currently, the synthesizable NoC components are generated using the Xpipes compiler [9], but our proposed framework is directly applicable to any other type of NoC architecture. The previous modules communicate using a common bus available in our FPGA board called On-chip Peripheral Bus (i.e. OPB in Figure 1).

Our emulation platform (see Figure 1) consists of four types of components: a control module, several types of *Traffic Generators* (TGs) or *Traffic Receptors* (TRs) and a device under test. The control module and the TGs/TRs are fully addressable by the processor for configuration and statistics acquisition purposes. Also, the control component can communicate with all the components of the platform by sending broadcast control signals to all of them. Each TG generates different traffic (see Subsection 4.1) and injects the packets into the *Device Under Test* (DUT) through its dedicated connections. Then, after passing through the network of switches, the traffic is received and analyzed by a set of TRs (see Subsection 4.2). Finally, to enable an efficient scalability in the amount of TGs/TRs, we have included in the platform a set of independent busses to connect them. Hence, using our architecture it is possible to plug up to 1024 TR/TG, assuming that a larger number of traffic devices would not be fit on actual FPGAs. As a result, this emulation platform enables to instantiate and to emulate real-life NoCs on current FPGAs.

In the following subsections we describe in detail the functionality of the main available components in our emulation platform (i.e. TGs/TRs and control module). We then detail the emulation flow of our system.

4.1. Traffic Generators

We define a Traffic Generator as a programmable module by a processor and controllable by a control module. To make it programmable, a bench of registers is addressable by the processor in each TG. Then, some control signals are used to communicate with its control module. Finally, an interface is available to inject traffic into the DUT. As this component must be able to explore/analyze many characteristics of NoCs implementations, the traffic generated by each TG is a function of the content of its registers. This feature gives us the possibility to generate different types of traffic with a single type of TG, and as the configuration of the registers of the TGs is done by the processor, we do not need to resynthesize the platform to perform different emulations of NoC traffic.

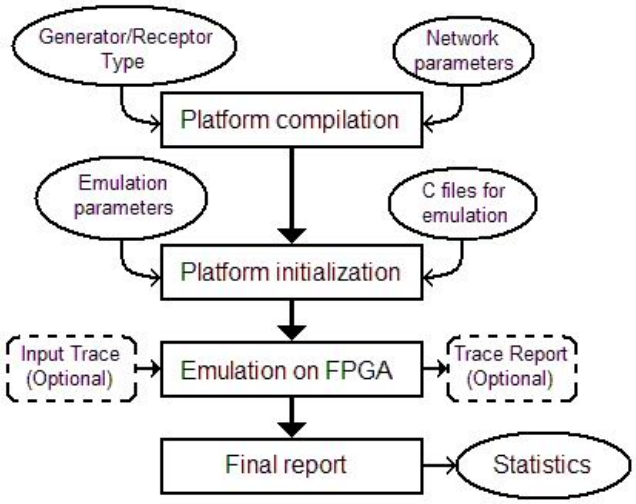


Figure 2. Our NoC Emulation Flow

In order to validate our approach, we have developed two types of TGs. The fist type can generate stochastic traffic. The second one uses input traffic traces generated by real-life applications, thus emulating the behavior of real workloads for NoCs (see [1] for more details).

Note that both types of TG can be used at the same time by including a controller for each used type.

4.2. Traffic Receptors

Similarly to the TGs, we have included two different implementations of TRs in our emulation platform. On the one hand, both have as common functionality the acknowledgment of the received traffic. In addition, both types enable two debug modes. In the first mode, it can perform an automatic check of the flits received via CRC check to guarantee that they are the correct ones sent by the TGs. In the second one, for manual checking, the content of the flits can be shown on the screen of the host PC to verify their content. The use of two different TRs supports an efficient implementation according to the required type of reports to generate and a suitable debug tool for the network.

Also note that the two types of TRs provide different kind of statistics to the user. The first type generates a histogram about the number of acknowledged flits. The second type generates a trace report for each received packet. The trace has the same format as the one used by the TGs. Hence, the processor can compute a detailed analysis (e.g. latency, arrival time) for each delivered packet.

4.3. Control Module

The control module is addressable by the processor and takes care of the synchronization of all traffic devices in the platform (i.e. TGs and TRs). For instance, it makes sure that all devices start the emulation at the same time. Also, the controller has the ability to reset the whole platform or even stop it, which is very useful if the emulation platform needs to be programmed to execute several consecutive emulations.

4.4. Emulation Flow

The main feature of our emulation framework and its flow is the simple initialization and statistics acquisition of any circuit level emulation without re-synthesizing and remapping the whole system. This is possible thanks to its mixed HW-SW structure, which allows the processor to initialize some parameters in the hardware part of the platform. An overview of this emulation flows is shown in Figure 2.

As Figure 2 indicates, from the hardware point of view, thanks to the components explained in Section 4 we can emulate at the circuit level various switching configurations of a NoC. The precise configuration to use in our emulations is defined in the first phase (first square in Figure 2) by configuring the Verilog code of our platform, which is a matter of defining several parameters.

After that, in our flow the initialization traffic to generate is performed (second box in Figure 2) using the processor of the FPGA board (i.e. Power PC). It executes a file with C code that contains the software code to configure the system. This file contains information about the total emulation time, the sampling period for statistics generation, routing information, latencies to use between packets, flits per packet and stochastic traffic distribution. Thus, this enables a high flexibility because no time-consuming recompilation of the HW involved is needed to emulate and study a wide range of these parameters. Then, during the initialization phase, by reading and writing in the TGs/TRs, the processor transmits data to TGs/TRs and is configured to receive results of the specified analysis at the end of the simulation.

After that, the emulation works autonomously and the TGs/TRs acquire the information necessary to generate the statistics demanded by the user in its C configuration file.

Finally, at the end of the emulation, the stored statistics are sent back to the processor which displays a summary report about the behavior and congestion of the network on the screen of the user using the monitor module (see Section 4) and its serial interface to the host PC. Instead of displaying the statistics on the user PC, the user can also program the processor to analyze it and perform a succession of emulation. We will see in Section 6 how this analysis can be perform.

5. Applications

In this section, we show two implementations of our emulation platform. A first implementation were shown in [1]. In this first implementation, the device under test was a network of switches. We then wanted to extend it by an implementation of the emulation platform by an emulation of a full Network-on-Chip, which include in addition to switches the network interfaces (NIs). In the next subsection, we describe both implementations.

5.1. Application 1: Emulation of a Network of Switches

The first implementation of our emulation framework emulates a Network of Switches. Those switches were generated by X-pipes compiler [9]. In the current version of the X-pipes compiler, the network protocol (i.e. the protocol used to communicate between switches and from NIs to switches) can be configured. There are currently 3 possibilities. The protocol, which we used, is called Ack/Nack protocol. This protocol includes a data link associated to a request wire, which indicates that the data link is valid. There are 3 additional wires, which acknowledge or not the request, and which indicate a repeated data in case of former non-acknowledgement. Having as a device under test a Network of Switches means that our TGs/TRs must have an interface compliant to this protocol.

In this implementation of the Emulation platform, we coded two kinds of TGs and two types of TRs. We can then describe those devices by showing two set of TG/TR. The first set is a stochastic TG associated to a TR, which measure the latency of packets and generate some histograms relative to the amount of traffic versus time. With a second set of TG/TR, instead of generating stochastic traffic, we generate traffic according to a trace. This implementation is more challenging since the trace (i.e. a collection of packet descriptors) must be sent to TGs in a continuous flow and must be decoded at run time by the TGs. Then, in this implementation, in addition to measuring the packet latency, we implemented a congestion counter, which measures the non-acknowledgement of flits thru the network of switches. The TRs also generate a trace according to the received traffic.

We show here an example of statistics collected with this emulation platform on Figure 3. We can see the average latency versus the number of packets per burst, with different burst length. This experiment shows how the latency of packets varies on a 2x3 mesh topology with a particular routing policy. The interesting result of this experiment is the speed to obtain such a plot. We have 15 points

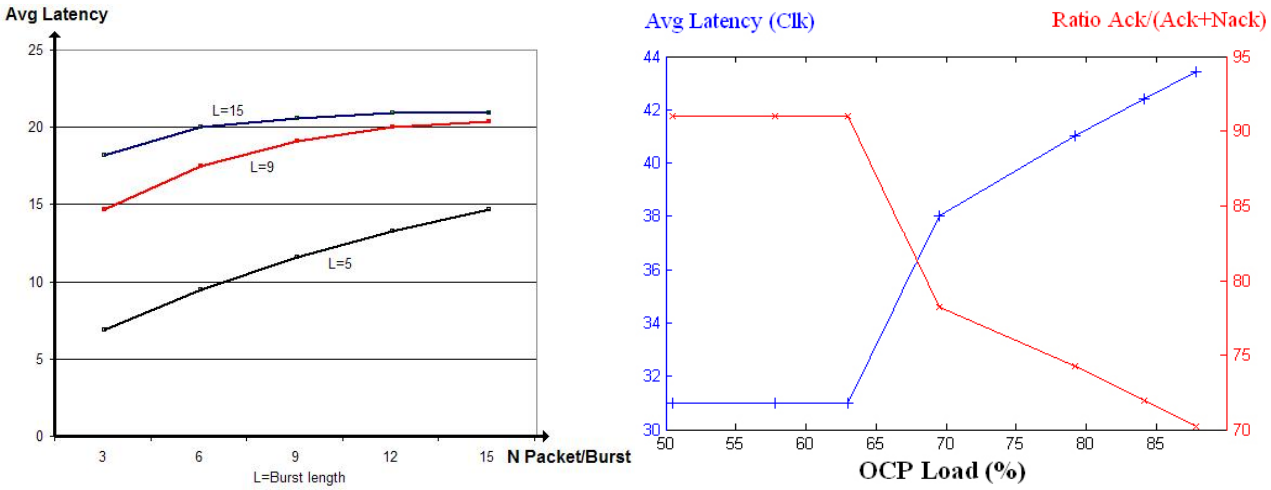


Figure 3. Emulation of a Network of Switches and a full NoC

on this plot. Using traditional simulator such a result is much longer to obtain (see [1] for more detail).

5.2. Application 2: Emulation of a Network-on-Chip

In this paper, we introduce for the first time a second implementation of our emulation platform. After emulating a network of switches, we wanted to emulate a full NoC. Adding some NIs associated to the network of switches implies a major change in the design of TGs/TRs. Instead of having modules, which generate or receive traffic, we must have devices, which emulate master or slave devices of a System on Chip. Those devices are controllable by the processor of the system like it was done in the previous design.

Unlike the previous design, we implemented in this case only a trace driven master programmable device. On the slave programmable side, we implemented a two state Markov chain, which turns on or off the device. We implemented this feature in order to be able to emulate the busyness of a slave device with a given probability. Note also that the protocol used is not the same as in the first application of our emulation platform since our Master and Slave modules communicates with NIs, which are OCP [22] compliant.

About generated statistics, the master device measures the average execution time of read and write operations. In the slave devices, we measure the average packet latency. We also added on the NoC links some sniffers, which measure the link activity. Note that our design is able to monitor the link activity without adding traffic on the network or without degrading the NoC performances. We insist on the fact that our emulation must not be intrusive in the design under test.

We are now showing now an experiment obtained with this emulation platform. It shows on Figure 3 a rising curve, which is the average latency. Also, it shows as a decreasing plot the ratio Ack/(Ack+Nack) on links of the NoC versus the OCP load. To perform this experiment, we have used a 2x2 mesh topology with attached to each switch, a couple of NIs, one with a master core, and one with a slave core. Then, we have generated traffic over the NoC, avoiding deadlocks and using all the links of the NoC. The average presented on this plot shows the results for all the links together and all the traffic. The interesting point of this experiment is, in addition to the speed of the emulation, the fact that we can clearly study the percentage of the OCP traffic generated by each master core. In this example, we can measure that the topology under test can accept up to 63 % of OCP load. This result is not surprising since we are using in our NoC links between switches of 16-bit width since the number of wires in the FPGA is limited. Thus, we could expect that this topology cannot ensure communication bandwidth at 100 %.

```
P_ini = {P_0 ini, .. , P_n-1 ini};
Do
    P_curr = P_ini;
    For(i=0;i<n;i++)
        For(j=0;j<P_i max;j+=parameter_step[i])
            Emulate topology;
        Extract best P_i;
        P_ini[i]=Best P_i;
    Compute distance D(P_curr;P_ini);
Until (D<"User defined maximum distance")
```

Simulation mode	Speed (cycles /sec)	Simulation time For 16 Mpackets	Simulation time For 1000 Mpackets
Verilog (ModelSim)	3.2K	13h53'	36 days 4h
SystemC (MPARM)	20K	2h13'	5 days 19h
Our Emulation	50M	3.2 sec	3'20''

Figure 4. Algorithm for NoC features exploration and emulation speed comparisons

6. NoC Features Exploration

To the best of our knowledge, our approach is the first one to introduce an exploratory algorithm at software level to extract the best topology for a given NoC application using FPGA emulation. In this section, we first explain all the steps of our algorithm, which is shown in Figure 4. Then, we present an example of application of this methodology. This example brings us to the comparison of the speed of our emulator with respect to traditional simulators.

Our algorithm is presented on Figure 4. It includes tuning of the most significant parameters of a NoC. For each parameter, we define a range of possible values. The fact is that exhaustive emulation of all possible topologies according to all possible values of the parameters is not affordable, even with an emulator, which runs at 50 MHz. In order to reduce the complexity of the algorithm, we instantiated an algorithm that consists of 3 loops, which varies one parameter at each time. Once a parameter has been tested under its range, we extract the best value.

To extract the best value of a parameter, we must first define the target of parameters tuning. Depending on the expected performance for the design we are able to extract the best value for all given parameters. In case several values are possible for the same overall result, the program chooses always the one closer to the first occurrence of the parameter value. After all the values for one parameter have been tested, we study another set of parameters. We compute the distance between the new set of parameters and the former set of parameters. If this distance is within a configurable threshold, we can consider that we have reached the best set of parameters possible. To compute the distance between two set of parameters, we can use the typical distance definition by computing the sum of the squared differences or it can be configured by the user

The complexity of this algorithm can be defined as follows: *Number of Parameters x Parameters granularity x Number of loops to reach the user max distance*. Even if this complexity is not as high as *Parameter granularity to the Number of Parameters*, it is still large. Therefore, to be able to execute that kind of algorithm, a fast exploratory mechanism, like the emulator we propose, is needed. In the following subsections, we show an example of use of our algorithm and we provide some key figures to show the gains in time achieved by our emulation platform.

We can take as example the definition and the tuning of NoC dynamic routing policies. In this case we can define a dynamic routing policy as follows: in NIs, if a path is congested, dynamically select another path. Then, the goal to achieve has to be set. In our case, we have decided to attain an effective utilization of NoC links without performance degradation. As a result, in this example the number of parameter is in the order of magnitude of 10 and the number of configuration to test with our algorithm is in the order of magnitude of 1000. Using a reasonable benchmark of 1 million cycles, we need to emulate one billion cycles, which requires more than the hundreds of KHz usually

achieved in cycle-accurate simulators [9,11,5]. In the next subsection, we evaluate the speed of our emulator compared to traditional cycle-accurate simulators.

On Figure 4, we compare the speed of traditional cycle accurate simulators with our proposed emulation framework. First, we note that simulators are running in the two case studies at 3.2 to 20 Kcycles per seconds, whereas our emulator runs at 50 MHz. Therefore, the emulation framework achieves speed-ups of up to 4 orders of magnitude faster than traditional simulators. Moreover, since forthcoming algorithms will require to tune ad-hoc NoC with real-inputs, long simulations/emulation for billions of cycles are needed. The result is that our emulator runs for one billion cycles in 3 minutes and 20 seconds while fast current cycle-accurate simulators would require more than 5 days.

7. Conclusions

New consumer products have increasingly higher demands and complex SoCs are used to implement such systems under the tight time-to-market constraints. NoCs solutions have been proposed to reduce the complexity of integrating tens of cores on-chip, but none of them allows complete architectural studies of different NoC realizations on silicon. In this paper, we have presented a flexible HW-SW emulation environment implemented on an FPGA that is suitable to explore, evaluate and compare at the physical level various custom NoC solutions for these new consumer systems with a very limited implementation effort. Moreover, as we have shown, a large set of important implementation and design parameters for actual NoCs can be evaluated on this proposed emulation platform in a very short interval, thanks to its HW-SW framework design to avoid multiple hardware synthesis on the FPGA and its fast emulation speed.

Acknowledgements

This work is partially supported by the Spanish Government Research Grant TIC2002/0750 and a Mobility Post-Doc Grant from UCM for David Atienza.

References

[1] N. Genko, et al. A Complete NoC Emulation Framework. In *Proc. DATE*, March, 2005.
[2] N. Genko, et al. A Novel Approach for NoC Emulation. In *Proc. ISCAS*, June, 2005.
[3] L. Benini, et al. Networks on chip: a new SoC paradigm. *IEEE Computer*, January, 2002.
[4] G. Brebner, et al. NoC with platform fpgas. In *Proc. FPT*, 2003.
[5] J. Chan et al. Nocgen: a template based reuse methodology for NoC. In *Proc. ICVLSI*, 2004.
[6] D. Ching, et al. Integrated modeling and generation of a reconfigurable NoC. In *Proc. IPDPS*, 2004.
[7] M. Coppola, et al. Occn: a NoC modeling and simulation framework. In *Proc. DATE*, 2004.
[8] W. Hang-Sheng, et al. Orion: a power-performance simulator for NoCs. In *Proc. MICRO*, 2002.
[9] A. Jalabert, et al. xpipescompiler: A tool for instantiating app. specific NoC. In *Proc. DATE*, 2004.
[10] S. Kolson, et al. A NoC architecture and design methodology. In *Proc. ASVLSI*, 2002.
[11] J. Madsen, et al. NoC modeling for system-level MPSoC simulation. In *Proc. RTSS*, 2003.
[12] T. Marescaux, et al. NoC as hw components of an os for reconfigurable systems. In *Proc. FPL*, 2003.
[13] F. Moraes, et al. Hermes: an infrastructure for low-area overhead NoC. *Integration-VLSI Journal*, 2004.
[14] S. Pasricha, et al. Fast exploration of bus-based communication architectures. In *DAC*, 2004.
[15] S. Pestana, et al. Cost-performance trade-offs in NoC: a simulation approach. In *Proc. DATE*, 2004.
[16] A. Pinto, et al. Efficient synth. NoC. In *Proc. ICCD*, 2003.
[17] D. Siguenza-Tortosa, et al. Vhdl-based simulation for Proteo noc. In *Proc. HLDVT Workshop*, 2002.
[18] L. Tang, et al. Algorithms and tools for NoC based system design. In *Proc. SBCCI*, 2003.
[19] D. Wiklund, et al. NoC simulations for benchmarking. In *Proc. IWSoC Real-Time Apps.*, 2004.
[20] C. Zeferino, et al. Rasoc: a router soft-core for NoC. In *Proc. DATE*, 2004.
[21] C. Zeferino,et al. Socin: a parametric and scalable NoC. In *Proc. SBCCI*, 2003.
[22] OCP International Partnership (OCP-IP). Open Core Protocol Standard. 2003. http://www.ocpip.org/home

Distributed Congestion Control for Packet Switched Networks on Chip

Théodore Marescaux[1], Anders Rångevall[1,2], Vincent Nollet[1], Andrei Bartic[1], Henk Corporaal[3]
[1]IMEC V.Z.W., Kapeldreef 75, 3001 Leuven, Belgium
[2]Department of Information Technology, Lund University, Lund, Sweden
[3]Technical University Eindhoven (TU/e), Eindhoven, The Netherlands
Theodore.Marescaux@imec.be

Packet-switched NoCs are efficient communication architectures for future MP-SoC platforms. However the run-time management of their communication, especially congestion avoidance is a challenging task. This paper discusses a distributed HW/SW congestion control technique. Hardware modules detect early signs of congestion and traffic is re-shaped accordingly to reduce congestion. The shape of the traffic is computed with binomial algorithms running on simple distributed microcontrollers.

The hardware and software extensions we propose to our system provide a congestion control solution that proves to be low-latency (100 clock cycles) and has a low area contribution to the router hardware (about 5% in UMC 0.13 technology).

1. Introduction

In order to meet the ever-increasing design complexity, many future sub-100nm platforms will be on chip multi-processor systems that require the flexibility and scalability of switched communication architectures such as Networks-on-Chip (NoCs). One interesting property of NoCs is their ability to provide a wide range of Quality-of-Service (QoS) levels, ranging from best-effort to soft and hard real-time guarantees.

We present the implementation of a packet-switched NoC and its network interfaces (NI) that provide soft real-time guarantees by performing traffic shaping. Traffic is shaped at the sender NI by controlling three parameters of the packets sent: their priority-level, their length and the time they are injected in the NoC. Adapting traffic shapes to the current quality requirements allows to control the congestion in the NoC and provides soft real-time guarantees. This paper focuses on the extensions of our NoC to allow an automatic and distributed mechanism to perform traffic shaping aiming at controlling congestion in the NoC. Our system -emulated on a Virtex-II FPGA coupled to a StrongArm processor- is composed of two NoCs: a 3x3 mesh NoC with virtual cut-through switching for application data traffic and a specialized NoC for control and distributed operating system (OS) services, such as controlling the traffic shaping parameters [1]. These OS services are implemented on small micro-controllers (μC) included in the NIs of the control NoC [2].

Enhancing our system with the ability to dynamically control the traffic shaping in a distributed manner has required a small number of changes, such as modification of the packet header, extension of the interface to the data router for observability of the buffer occupancy and control algorithms implemented in software.

Related work comes in several categories: some authors propose to enforce hard real-time guarantees thus completely avoiding congestion but at the expense of more complex and restrictive scheduling techniques [3]. Other techniques use adaptive routing to distribute traffic and avoid congestion but need to either re-order packets at the expense of large buffers in the network interfaces or need to re-send dropped packets and hence incur a large latency penalty [4,5] . Congestion avoidance by central control of traffic shaping is discussed in [2] and centralized end-to-end flow control in [6].

2. NoC Extensions

Providing distributed congestion control on the data NoC requires hardware and software modifications to our system. The packet header has been extended with the address of the source of the communication, to be able to identify the source of the congestion, and with a priority field (Figure 1). The router has not been modified since it processes the contents of the first header flit and the header extensions only affect the second header flit. The FIFOs inside the data NoC routers have been modified to provide visibility on the number of packets queued and on the *source* and *priority* fields from the header of the last packet buffered. The resulting signals are fed into a *buffer monitor* module that has the role of sensing congestion. A FIFO is considered congested when its occupancy goes above a settable threshold level. The *buffer monitor* keeps a history of *source* and *priority* of the latest packets received. Upon congestion, the history is searched and the *source* of the lowest priority packet is notified to a *congestion monitor* module. The *congestion monitor* multiplexes the congestion notifications of all *buffer monitors* in the router and copies them to the data memory of the μC in the local control NI. The μC local to the router that has sensed congestion can then notify the μC of the *source* router to throttle the traffic coming out of its data NI. As a consequence, congestion is reduced and the path is left free for other communication streams sharing part of the congested path.

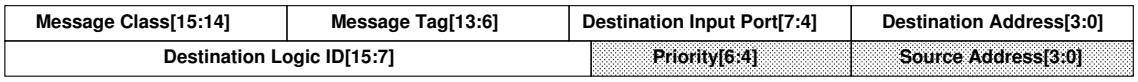


Figure 1. Packet header extended with source and priority fields.

3. Buffer Monitor

The relatively small size of the buffers in the routers of the data NoC leads to a network that can congest rapidly. In worst-case scenarios, congestion can build-up in as little as $3 * buffer\_depth$ clock cycles, hence the congestion detection mechanism needs to rapidly counteract. The congestion detectors are called *buffer monitors* and there is one *buffer monitor* connected to each output buffer in the routers of the data NoC. On a 2D mesh NoCs a router has between three to five buffers and hence as many *buffer monitors*.

The *buffer monitor* uses the four signals that are exported from each buffer in the Data Router to get information about the packets (Figure 2). Whenever a packet enters the buffer, the router asserts the signal `new_packet`. At the same time the router exports the source and priority of that packet, the `source` and `priority` signals. The fourth signal from the router is the `packets_outqueue` that tells how many packets there are in the buffer.

The primary role of the *buffer monitor* is to sense congestion by measuring how many packets are stored inside the buffer in the router. If there are more packets than the `threshold` level set by the operating system, the buffer is considered to be congested.

Upon congestion, the role of the *buffer monitor* is to select where to send a congestion notification. To make an informed decision, the *buffer monitor* stores information regarding recent packets

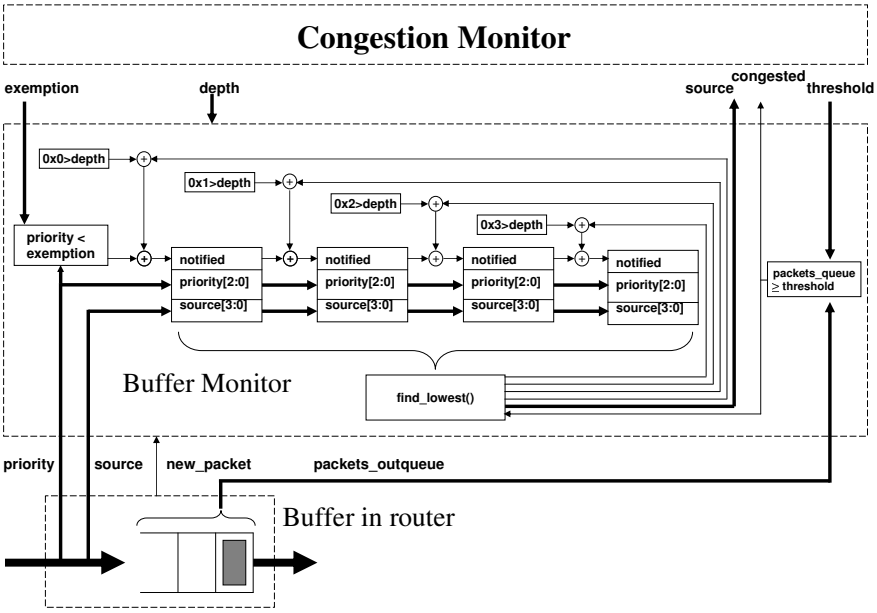


Figure 2. The *buffer monitor*, keeps a history of packets sorted by priority. Upon congestion it reports the lowest priority packet to the *congestion monitor*.

in a separate history FIFO (hFIFO) that contains the source address and the priority of a packet. Whenever a packet header enters the data router buffer, its source address and priority are shifted into the hFIFO.

When a *buffer monitor* senses congestion it uses the `find_lowest` function to search among all the priority fields inside the hFIFO to find the source address of the packet that has the lowest priority. If two packets have the same priority the most recent packet is selected.

A congestion notification containing source address and priority is then reported to the *congestion monitor* (Figure 2). The entry in the history FIFO is then tagged as `notified` to avoid subsequent congestion notifications. The `notified` tag is local to the *buffer monitor* hFIFO, therefore the same packet may trigger more congestion notifications on the next hops along the path.

The maximum size of the hFIFO is set at design-time by the `HISTORY_DEPTH` constant. Nevertheless the effective size of the hFIFO can be controlled at run-time, for the needs of the control algorithm, by setting the `depth` register in the *buffer monitor*. All entries in the hFIFO are searched in parallel with a search tree to allow a very short reaction time, at the expense of increased area. However that area increase is only linear with `HISTORY_DEPTH` (Section 7). The hFIFO implemented in our emulation platform stores information about four packets.

Two `depth` settings are of particular interest: 0x0 and $buffer\_depth$. With a depth of 0x0, information is only stored for one packet. This packet is the most recent one that has entered the buffer. This setting is interesting because it equates to very simple hardware, just store and search one packet. The other setting where `depth` equals $buffer\_depth$ means that the hFIFO stores information about as many packets as the actual buffer holds.

The threshold and depth variables, although they seem similar, are quite different. The threshold variable defines when the buffer is considered to be congested. This is done by comparing the

threshold value to the number of packets in the buffer. The depth variable on the other hand only affects for how many packets the source address is remembered.

Another setting of interest is when the `threshold` and `depth` are both set to the same number of packets. For example if the threshold and the depth are both set to two packets, then the source address will be taken from one of the two currently buffered packets. It will not be taken from a packet that has already left the buffer, which can be the case if the threshold is set to two packets and the depth is set to three packets.

To further enhance the flexibility of the congestion control, there is a possibility to completely exempt packets with certain priorities from triggering a state of congestion. This behavior is controlled through the `exemption` register. By setting the `exemption` to 0x3, packets with a higher priority than three – and cause congestion – will never trigger any congestion notifications to be sent. When congestion is caused by another packet, the tile that sends the exempt packet will never receive a congestion notification. In this example the packets with priorities 0x2, 0x1 and 0x0 cannot trigger any congestion notification from being sent. It is important to realize that the exemption mechanism does not prevent congestion from occurring; it merely hides it from the congestion control.

4. Congestion Monitor

Every router on the data NoC contains one *congestion monitor* that fulfills two major roles. On the one hand it acts as a multiplexer/decoder to give the local μC visibility and control over all the *buffer monitors* contained in the data router. On the other hand it is able to autonomously send/receive notification of congestions to/from remote *congestion monitors*. Congestion notifications are sent over a separate control network reserved for low-latency OS traffic [1,2].

Each *congestion monitor* connects to a μC in the local Control Network Interface (Figure 3) and its registers are memory-mapped in the address space of the microcontroller. It is the microcontroller that assigns values to the different variables such as the `depth` and the `threshold` that affect the *buffer monitors*. The *congestion monitor* also contains statistics registers that count the number of congestion notifications received and sent.

The μC controls the behavior of the *congestion monitor* and of its *buffer monitors*. For instance it can selectively enable *buffer monitors* through the memory-mapped register `cmBufMsk`. Incidentally the *congestion monitor* also contains a hardware timer that is read and set by the μC. It is used to provide the μC with a time-base related to the NoC traffic.

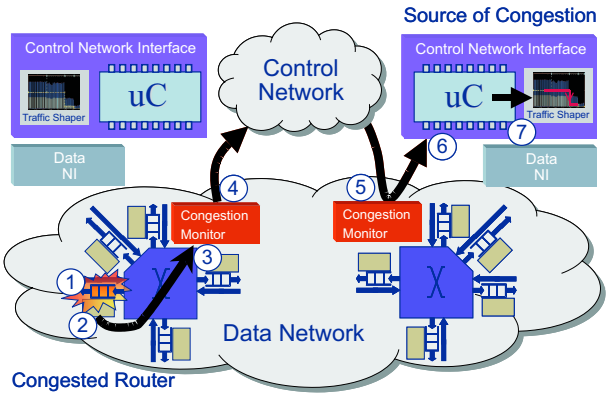
Congestion signaling of the *buffer monitors* is detected by the *congestion monitor* and is relayed as a congestion notification. The destination for the congestion notification is provided by the *buffer monitor* signal `source`. If several *buffer monitors* of the same router indicate congestion at the same clock cycle, only the highest priority congestion notification is sent, the other ones are dropped.

Congestion notifications can either be directly sent from congestion monitor to congestion monitor through the control network, or the autonomous congestion notification can be overridden by the μC by setting the `autonomous_mode` register to zero in the *congestion monitor*. The latter allows the μC to implement in software more complex notification schemes but puts it in charge of sending congestion notifications. By default our system functions in autonomous mode.

5. Congestion Control in Action - an Example

This section shows a simple example of how congestion is detected and how congestion notifications are propagated to the source of congestion. The congestion control mechanism is triggered when a buffer in a router on the data NoC fills up above the threshold level and is thus considered as

being congested. Figure 3 shows how the congestion notification is propagated through the system. Steps one to five take place in hardware and take three clock cycles to execute (not including possible, but short, conflicts on the control network). Steps six and seven execute in software on the local μC, typically consuming about 100 clock cycles.



1. The *buffer monitor* detects congestion in the local buffer.
2. The *buffer monitor* searches packet history for congestion source.
3. The source address of the congestion is given to the *congestion monitor*.
4. The *congestion monitor* sends a congestion notification to the congestion source through the *control network*.
5. The congestion notification is processed by the receiving *congestion monitor*.
6. The *uC* polls the *congestion monitor* and gets the congestion notification.
7. The *uC* computes a new window size and configures the *traffic shaper* to reduce throughput accordingly.

Figure 3. Congestion control in action. A buffer in the left router is congested. The arrows show how the congestion notification is propagated to the source of congestion.

6. Traffic Shaping - Control Algorithm

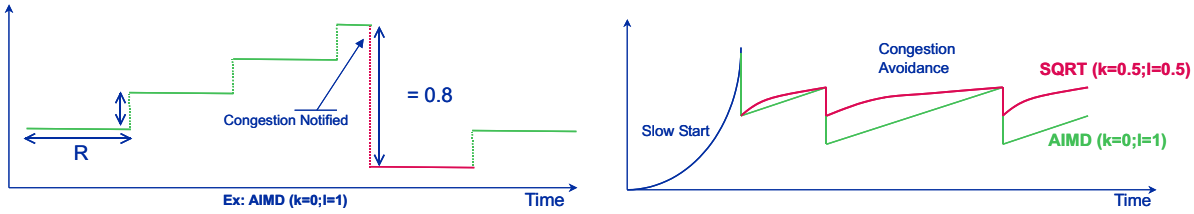
The traffic shaping in our system is based on a sliding window mechanism. Packets are only injected in the network during the time a window ω is opened. The size of the window is based on binomial congestion avoidance [7]:

$$Window\ Increase: \quad \omega(t+R) = \omega(t) + \frac{\alpha}{\omega(t)^k} \quad ;(\alpha > 1) \tag{1}$$

$$Window\ Decrease: \quad \omega(t+\delta t) = \omega(t) - \beta\omega(t)^l \quad ;(0 < \beta < 1) \tag{2}$$

While no congestion is notified, the window size is gradually increased at a rate R (Figure 4(a)), by default 2048 NoC cycles on our system. In the general case, the increase amount is proportional to ω^{-k} (Equation (1)). Shortly after congestion has been notified, the window size is decreased (Equation (2)), usually by a larger amount than it is increased (Figure 4(a,b)). The parameters k and l in Equations (1,2) define the aggressiveness at which the windows are opened and closed and therefore their impact on response to congestion. To ensure a good trade-off between probing aggressiveness and congestion responsiveness, we use the $k + l$ rule defined in [7]: $k + l = 1$ and $l \leq 1$. Figure 4(b) shows the effect of two different sets of (k, l) values that follow the $k + l$ rule. Additive Increase Multiplicative Decrease (AIMD) uses $(k, l) = (0, 1)$ and yields a windowing mechanism that is both efficient and simple to implement. The square root algorithm (SQRT) uses $(k, l) = (0.5, 0.5)$ and thus changes the window size proportionally to $\sqrt{\omega}$ which yields a smoother traffic shaping but is more computationally intensive (Figure 4(b)).

The μCs in the network interfaces are tiny micro-controllers, without floating-point units and have a very limited amount of memory, making it difficult to store constant tables. Therefore, to



(a) Window increase/decrease illustrated on AIMD. (b) SQRT gives a smoother window variation.

Figure 4. Traffic shaping is based on a sliding-window mechanism.

rapidly compute window sizes, the congestion control algorithm initially implemented on our system is AIMD. In the current implementation the sending rate is divided by half upon notification of congestion and the increase rate is 2048 NoC cycles, or the equivalent of sending 8 packets of maximum length (512 bytes).

7. Results

The distributed congestion-control system is fully implemented and integrated to our emulation platform. An instance with a history depth of 3 packets and 8 levels of priority requires about 3% of the Virtex-II 6000 for all 9 congestion monitors and 33 buffer monitors. The average total response time to congestion is 100 NoC cycles with a variance of 20.

7.1. Synthesis to UMC 0.13 STD Cell Technology

Additionally to the FPGA implementation, we have also synthesized the congestion detection system to standard cell technology using the UMC 0.13 libraries. We believe these numbers to give a good indication of the impact our distributed congestion control system would have in a real chip.

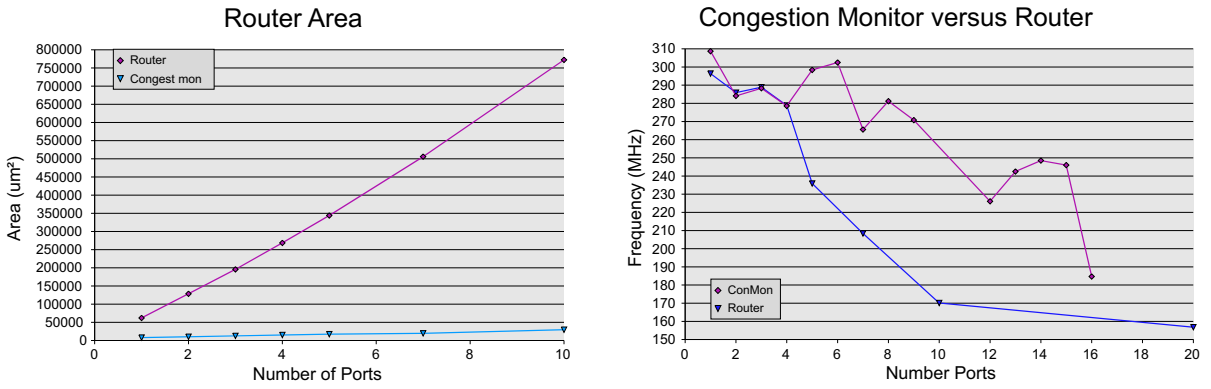
We have compared the area and contribution to the critical path of the whole congestion controller -composed of a congestion monitor and 4 buffer monitors- to the router of the data NoC. It results that the area impact of our congestion control system is negligible with respect to the router area (Figure 5(a)). For instance for a router with 4 ports, the congestion controller occupies about $15000\mu m^2$, which only represent 5% of the router. As Figure 5(b) shows, the critical path of the congestion controller has a negligible effect on that of the router.

Figure 6(a) shows a linear increase to the area contribution of the congestion controller when varying the number of ports, hence the number of *buffer monitors*. Figure 6(b) details the area contribution of a congestion controller with only one *buffer monitors* when varying its `HISTORY_DEPTH`. The dependency is again linear, which shows that `HISTORY_DEPTH` can be increased to slightly higher values to allow more complex control algorithms without having a dramatic impact on area.

7.2. System Latency

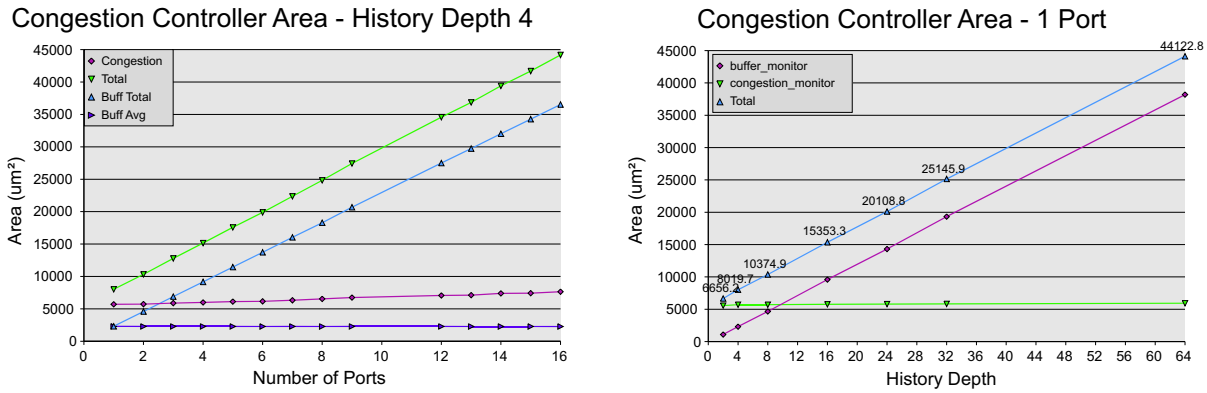
A critical requirement of our system is a low latency when responding to a congestion notification. We measure that response latency from the moment a buffer in a data NoC router is congested to the moment the window size is adapted in the traffic shaper to reduce the data throughput at the source of congestion (steps 1 to 7 in Figure 3).

The polling by the microcontrollers determines to a large extent the latency for the congestion control. The whole control loop executes in about 100 clock cycles (Figure 7(a,b)), unless it needs to respond to commands sent from the operating system.



(a) Negligible area of congestion controller. (b) No impact on router critical path.

Figure 5. The congestion controller proves to be inexpensive (UMC 0.13 STD cell technology).

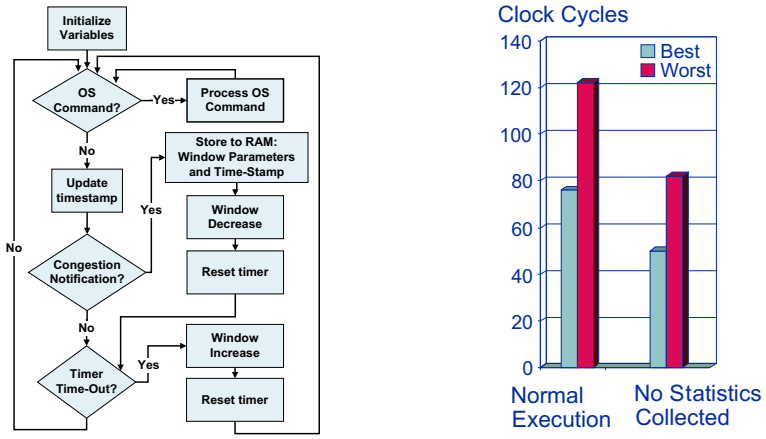


(a) Congestion controller area vs buffer monitors. (b) Congestion controller area vs history depth.

Figure 6. Congestion controller implementation details in UMC 0.13 STD cell technology.

The hardware composed of *buffer monitors* and *congestion monitors* is optimized, so that the congestion detection and notification only takes 3 clock cycles. All the remaining cycles reported in Figure 7(b) are spent in software on the μC local to the source of congestion to compute window sizes with AIMD, perform I/O accesses or respond to requests from the central operating system. With statistics collection and reporting enabled, the average response latency to congestion is about 100 clock cycles, with a variance of 20.

On our system the packet payload length can vary between 0 and 256 data words (16-bits). The packet travel time is proportional to the number of hops in the system and to the payload length. For packets longer than 100 data words, the congestion control system reacts in a time that is inferior to the packet travel time. For more reasonable packet lengths, the latency of the congestion control is in the order of magnitude of the flight time of 10 packets. The system is therefore fast enough to rapidly counteract when congestion is building-up in the NoC.



(a) Control loop on μC. (b) Reaction latency.

Figure 7. Control loop on the μC and impact on congestion control latency.

8. Conclusion

This paper discusses distributed congestion control for packet-switched networks-on-chip. Early signs of congestion are detected by distributed hardware modules and traffic re-shaping techniques are used to reduce congestion accordingly. The traffic shaping is based on a sliding-window mechanism for which the window size is computed by software running on distributed microcontrollers that implement simple operating system support on our platform.

The hardware and software extensions we propose to our system provide a low-latency, low-area congestion control solution. The area overhead of the hardware extensions is negligible compared to the area of the packet-switched router (only 5% of the router area in UMC 0.13 technology). The latency of the congestion control system is about 100 clock cycles which is low enough to react to congestion building-up in the system.

Both hardware and software extensions are very flexible and parameterizable to allow the implementation of many different types of control algorithms. It is possible to change the parameters and algorithms at run-time from the central operating-system on the platform. Future work will focus on determining optimal parameters and control algorithms depending on traffic patterns.

References

[1] T. Marescaux, V. Nollet and al., ”Run-time support for heterogeneous multitasking on reconfigurable SoCs”, in Integration VLSI Journal, Elesvier Science Publishers, 2004.

[2] V. Nollet, T. Marescaux, D. Verkest, J-Y. Mignolet, S. Vernalde: ”Operating System controlled Network-on-Chip”, Proc. of the Design Automation Conference, San Diego, June 2004, pp. 256-259.

[3] K. Goossens, J. Dielissen, and A, Rădulescu, ”The Æthereal Network on Chip: Concepts, Architectures, and Implementations”, in IEEE Design and Test of Computers, September 2005.

[4] A. Adriahantenaina et al., ”SPIN: A Scalable, Packet Switched, On-Chip Micro-Network”, DATE Conference, 2003.

[5] M. Millberg, E. Nilsson, R. Thid, S. Kumar, and A. Jantsch, ”The Nostrum backbone - a communication protocol stack for networks on chip”, in Proceedings of the VLSI Design Conference, January 2004.

[6] P. Avasare, V. Nollet, J.-Y. Mignolet, D. Verkest, H. Corporaal, ”Centralized End-to-End Flow Control in a Best-Effort Network-on-Chip”, in Proceedings of EMSOFT, September 2005.

[7] D. Bansal and H. Balakrishna, ”Binomial Congestion Control Algorithms”, INFOCOM 2001.

Versatile FPGA-Based Functional Validation Framework for Networks-on-Chip Interconnections Designs

Javier B. Perez Ramas[⋆†], David Atienza[⋆‡], Miguel Peon[⋆], Ivan Magan[⋆], Jose M. Mendias[⋆], Roman Hermida[⋆]
[⋆]DACYA/UCM, C/ Prof. Jose Garcia Santesmases s/n, 28040 Madrid, Spain. {datienza, mendias, rhermida}@dacya.ucm.es ; {mikepeon}@fdi.ucm.es
[†]Working for INDRA Sistemas, S.A. Avda. de Bruselas 35, 28108 Alcobendas, Madrid, Spain. jbpramas@indra.es,
[‡]LSI/EPFL, EPFL-IC-ISIM-LSI Station 14, 1015 Lausanne, Switzerland. david.atienza@epfl.ch

Forthcoming System-On-Chip (SoC) devices will be made of hundreds of cores to be able to handle the complex functionality required in new consumer devices. In the last years, Network-On-Chip (NoC) has been proposed as a promising replacement for buses and dedicated interconnections to solve the scalability and complexity problems. As a result, NoCs imply a complex design process to define suitable network interfaces to access the on-chip network, the selection of convenient protocols and also topologies of switches to transport the data. Presently, several options for NoC topologies and interfaces have already been proposed at different levels of abstraction, but extensive testing is still required to functionally validate them at the physical level in environments with a variable number of processing cores and real applications. In this paper we illustrate the use of reconfigurable hardware (FPGA) to help bridging the gap between NoC design and validation. The defined reconfigurable system has enough memory capabilities to run complex SoC applications and can use any number of processing cores. We assess its effectiveness for NoC-based exploration with two case studies: two different applications running in the same SoC, which is connected using a standard NoC and a commercial bus solution.

1. Introduction

Recent research claims that the interconnection model inside a chip, traditionally based on a bus, can become a bottleneck in the near future due to hundreds of IPs with many simultaneous communications between them [1,6] causing permanent conflicts in the shared medium. More complex arbitration protocols, aggressive hierarchization and segmentation of the buses could be necessary to minimize those effects. Physical implementation is difficult because of the huge size of the shared structures connected to an equally high number of end-points. At system level, buses have been displaced by packet-based interconnection networks (standards as RapidIO, VME-X or PCI Express replace the traditional buses like VME64 or PCI gradually). The advantages of this option have already been studied: speed, scalability, reliability, resistance to the congestion and many others outlined by its defenders [12]. As an extrapolation of the previous idea within the scope of Systems-on-Chip (SoCs), a technology baptized as Network-on-Chip (NoC) has been recently proposed. A NoC implements a commutation network inside the chip in order to communicate the different Intellectual Property blocks (IPs from now on). The objectives are multiple, such as, increasing the efficiency of the IP communications, allowing multiple communications at the same time and, in general, solving the bus bottleneck [1,12].

Diverse NoC architectures have appeared and numerous theoretical studies indicate the advantages of this option. Although attempts of real tests for these systems exist (see Section 2), there have not been so far complete real-life tests that compare designs of real and customizable SoCs with standard

applications, and where different interconnection alternatives are utilized. Obtaining exact measures of real life applications with the different alternatives must be possible. In addition, the applications should be as much as possible interconnection-network independent, so that to prove the advantages of a new option it is not necessary to rewrite the function calls or mechanisms to interact with the intercommunication layer. Nevertheless, the eventual software to tune to exploit all the benefits of such intercommunication (e.g. message passing vs shared memory). Our platform tries to offer all of these characteristics.

The rest of this paper is organised as follows. First of all, in Section 2 a summary of relevant research in NoC and bus-based analysis and prototyping is presented. Then, in Section 3 we detail the architecture of our proposed Multi-Processor SoC (MPSoC) platform. Next, in Section 4 we describe two study cases, one for buses and another one for a NoC-based design and indicate the synthesis results obtained. Later, in Section 5 we describe two different real-life parallel applications running in our MPSoC platform and the empirical results obtained. Finally, in Section 6 we present our conclusions and possible future research lines.

2. Related Work

Recent studies propose that the bottleneck present in the traditional bus interconnection system can be solved for the SoC domain using the Network-on-Chip approach so a significant effort has been made to provide functional NoC architectures. [14] presents Xpipes, a NoC development framework highly parametrizable and flexible enough for investigatation due to the possibility of experimenting with customizable network topologies. Also, conversely to other precedents like Aethereal [15] or Proteo [10], Xpipes is the first technology presented with a compiler of NoC descriptions that allows to adapt the topologies and characteristics of the NoC for each application. Xpipes provides a complete and flexible development system of NoC with a library of personalizable components, such as, Open Core Protocol (OCP) [9] Network Interfaces (NIs), switches and links, and a compiler: XpipesCompiler [5]. The input is a file with the topological and parametric description of the desired NoC and the output is a SystemC file containing the implementation. Also, in [8] the interesting Hermes framework is explained. It is a NoC with a simple switch, which is economic in area, and with implicit routing in the destination address.

In addition, previous work using FPGA for NoC-based research exists. In [7] and [2], NoCs with a mesh-based topology and packet-switching as communication mechanism have shown the effectiveness of NoC. Also, in [4] a Xilinx Virtex 2 Pro FPGA is used to explore the performance of NoC solutions using Xpipes. However, it is limited to synthetic applications through traffic generators or previously generated traces. Finally, several NoC architectures (e.g. torus) and designs of switches/routers have been ported to FPGAs to validate their features (e.g. packet sizes, switching-mode) decided by additional HDL simulations [7,11,13,3]. None of this works enables a complete flexibility for testing interconnection mechanisms with real-applications as we propose in this paper with our FPGA-based MPSoC platform.

3. Proposed MPSoC Platform

The main characteristics of our generic MPSoC platform include a variable number of general purpose CPUs, standard connection that allow attaching any IP, the possibility to define or use any interconnection topology and the visibility of all the architectonic issues of the system in order to evaluate the interconnection, either in simulation or during real-time execution. The platform has been developed using a Virtex 2 PRO 20 FPGA from Xilinx [16], a suitable environment because it

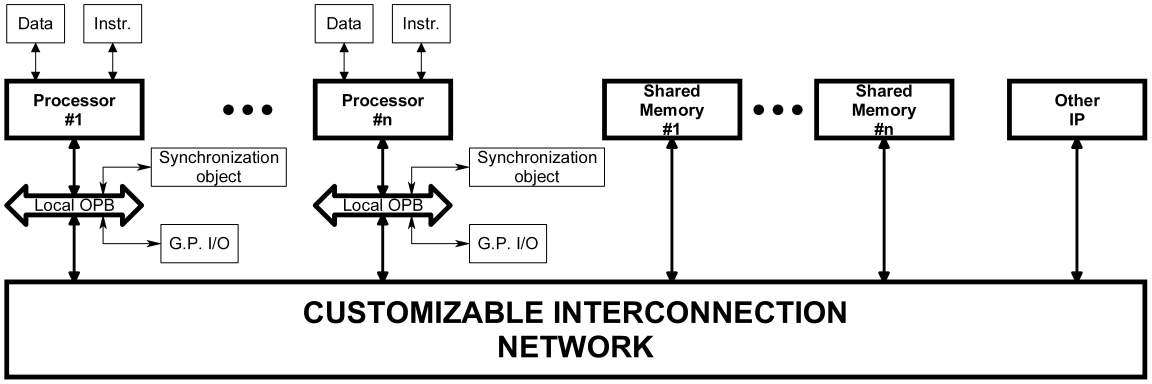


Figure 1. Generic architecture

provides a very high-density reconfigurable hardware platform, different processing cores (i.e. hard-wired PowerPCs and soft-core MicroBlazes), SRAM memories and various interconnection buses, such as, On-chip Pheripheral Bus (OPB) or Processor Local Bus (PLB).

The generic architecture of the system mapped onto the FPGA is shown in Fig 1. As it shows, its main components are the following ones:

- Processors: The general-purpose processors used are Xilinx MicroBlaze CPUs [17], a synthesizable VHDL 32-bit CPU based on a Harvard architecture. MicroBlaze uses local memory buses for instruction and data, and standard IBM OPB-bus external connection. Each CPU has its own private data and instruction memory.

- Interconnection network: The interconnection is the element under test and enables data transmission between the different IPs. Its choice is configurable to NoC or buses and affects the system behaviour. Moreover, the same NoC technology can be configured for different topologies, packet routing, etc. In all cases the network must provide an OPB interface to be connected to the platform.

- Shared memories: Internal FPGA memory is currently used, although external memories (e.g. SDRAM, DDR, etc.) can be easily added via specific controllers.

- Other IP cores: Any module (generic I/O, synchronization device, etc.) that uses the standard OPB interconnection or the OCP interface can be directly added to the SoC and connected to the interconnection used.

3.1. HW/SW MPSoC Toolflow

One of the most important elements in our MPSoC platform is the inclusion of the complete HW and SW flows in one overall toolflow. An overview of this working flow is presented in Figure 2. On the one hand, regarding the HW part, the general architecture of the system is given in the form of a Xilinx Embedded Development Kit (EDK) description. We provide the necessary IP cores in EDK format and the templates with the system interconnection and external interfaces. EDK is also used to instantiate the desired number of MicroBlaze processors. On the other hand, related to the SW part, the standard GNU GCC is used to compile the sources for the processing cores. At this point, all the elements of the SoC, except for the interconnection network, are integrated. The external ports of the generated EDIF contain the nets necessary to attach the interconnection network.

As interconnection network we have used the Xpipes NoC [14]. In our case, XpipesCompiler takes as input a file with the topological and parametric description of the desired NoC and produces

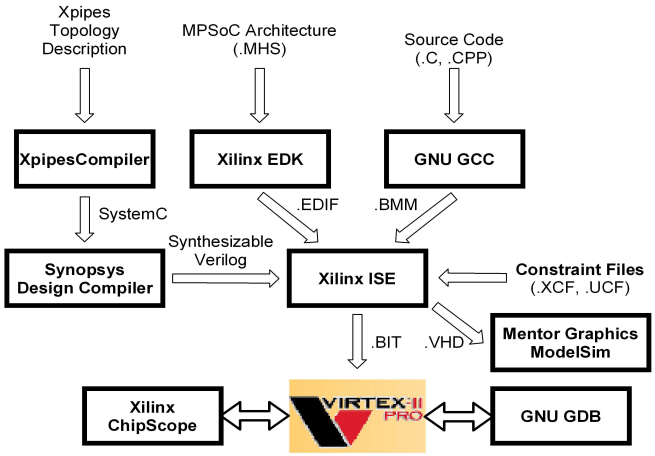


Figure 2. Working flow for the NoC case

as output a SystemC file containing the implementation. Therefore, Xpipes can be used and adapted for a wide range of applications. Using the Synopsys Design Compiler [18] synthesizable Verilog files are obtained from the SystemC ones. Xpipes uses the OCP protocol for interconnection of the network interfaces with the external cores; Thus, we have developed an OPB-to-OCP bridge in order to connect the MicroBlaze cores to it.

Then, we use Xilinx ISE to mix the obtained EDIF with the interconnection network Verilog files and the platform description VHDL ones. Finally, Xilinx tools map the whole design onto the final FPGA and the architecture with the application sources is ready to be downloaded onto the chip to start the testing process.

In addition, our proposed flow enables the use of additional debugging or simulation tools. Extensive testing and architecture analysis can be done at design level with Mentor Graphics ModelSim. Furthermore, at run-time, Xilinx ChipScope and GNU GDB can be used to debug and trace the framework. Statistics of the network can be obtained either via simulation or real execution on the platform.

4. Case Studies

To demonstrate the use of the MPSoC framework we present two case studies. First, a general purpose MPSoC system connected by a hierarchy of buses and, second, we have replaced the buses by a Xpipes-based NoC.

4.1. Baseline MPSoC Configuration

The test system fits suitably in a Virtex 2 PRO 20 FPGA with both interconnection options. We use four MicroBlaze processors with 4 Kb of private memory and two 64-Kb shared memories implemented in the FPGA. All memory accesses out of the shared range are immediately dispatched in the private ambit of the CPU and do not use the interconnection network. Interconnection bridges, depending on the chosen implementation, route the shared memory accesses and locks the local OPB until they are resolved.

4.2. Case study 1: bus based

First we show an interconnection based upon a simple bus hierarchy, using the standard OPB: a synchronous, master-slave, standard IP bus that can be efficiently implemented on a FPGA and is directly compatible with MicroBlaze. The bus hierarchy allows the CPUs to access the shared memories. There are as many OPB private buses as processors, and an additional public bus. Each

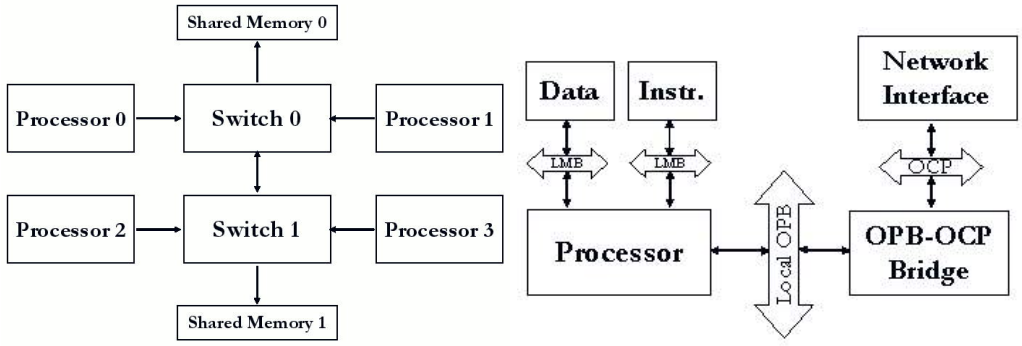
Table 1
Use of resources with both interconnection alternatives

Platform version	Slices
Bus based	3,050 of 13,700 (22%)
Xpipes NoC based	9,497 of 13,700 (69%)

private OPB has an OPB-OPB bridge connected to the shared medium that has also the shared RAM. Each OPB bus has its own arbiter, which has a fundamental role in the behaviour of the system. In all cases, we use the Xilinx dynamic arbiter [19] with a LRU priority system.

4.3. Case study 2: Xpipes NoC based

For this case study we have instantiated a custom 2 switches NoC-based interconnection topology using XpipesCompiler. The overall system architecture is shown in Figure 3.



(a) Global view (b) Detail of processor connection

Figure 3. Implemented NoC topology

In order to be able to connect the MicroBlaze using the structure explained in the previous point, it is precise to establish a connection between the OPB of the CPUs and the corresponding Network Interface (NI). As the NI implements a standard OCP interface, a bridge between OPB and OCP can serve to this purpose. In the present work a specialized specific module has been created for this purpose, written in VHDL. This module works as slave in OPB and as master in the OCP side. Optionally, the peripheral can implement a prefetch buffer of customizable size. This buffer anticipates the reading of contiguous data and does not have effect in the writings. Writes and prefetched reads have minimum latency. This has been done to minimize the impact of the NoC intrinsic latency and to take advantage of burst transfers functionality of Xpipes.

4.4. Synthesis Results

The use of FPGA resources for each version is shown in Table 1. Note that the NoC version is more than triple the size of the bus one. One first conclusion of this study is that the NoC generated by XpipesCompiler is optimized for silicon, but it is not the most suitable one for the internal structure of the FPGA architectures. Similar results can be envisaged for other NoCs with complex internal switches and NIs.

Table 2
Overall results obtained in Case Study 1

	Bus	NoC, prefetching (2 words)	NoC, no prefetching
T (cycles)	1.759×10^{6}	1.716×10^{6}	$1{,}925\times10^{6}$
BW (bytes/cycle)	0.149	0.153	0.136
BW/slices	4.88×10^{-5}	1.61×10^{-5}	1.43×10^{-5}

5. Applications and Empirical Results

In order to test the platform and to show its versatility to compare different interconnection strategies we show two test applications that run on both interconnection options (bus and Xpipes NoC). In the case of the NoC case study, tests have been done with both prefechting enabled and disabled. Both applications are based on image processing with graphic frames stored in the shared memories. For each case, we measure the total execution time (T in Table 2 and Table 3). Because our experiments pursue architectural results, all data collected is expressed in clock cycles and not in real time.

Also, it is interesting to provide a measurement of the performance of each option and relate it to its architectural complexity. Bus is likely to have less performance than the NoC option but NoC uses more area, as some authors outline in their studies [5,7,8,11]. To approximate this relation, we calculate an efficiency coefficient as the quotient between the effective bandwidth of the application (BW in Table 2 and Table 3) and the number of slices that the implementation occupies.

5.1. Parallel greyscale dither

In this application, a 256x256 pixels grayscale image is stored in each shared memory. Each pair of processors work in one image to obtain the monochrome dithered image and store it in the same memory.

The experimental results (see Table 2) show that the application performance is not deteriorated by the use of a shared bus. The reason is that regular algorithms are executed across all the processors. Therefore, when an algorithm accesses the memory in a regular way, and the processing time gives enough margin for it, the accesses tend to align themselves after the initial conflicts. Hence, some processors access the memory while others process the last fetched data. As processing and access times are almost constant, once balance is reached, this situation is preserved for the rest of execution.

In the case of NoC without prefetching, the application is significantly slower. This is due to the high number of readings, as many as writings, which suffer from the extremely high latency of the NoC for single read-requests. Latency is almost insignificant for writings, but it is still not enough to accelerate the execution of the algorithm. A moderate improvement occurs with prefecthing. In this case, the execution takes around 11% less time compared to the NoC version without prefetching, and it is even slightly faster than the bus one. This is because each pair of CPUs can access its shared memories without colliding with the other pair. Particular reasoning deserve the results of the efficiency coefficient, as it is more than three times better for the bus than for the NoC: in order to obtain a performance improvement of around 2.5% with respect to the bus version, the NoC requires more than three times of space in the FPGA. From these results we can conclude that the bus, in this case, is the option that presents the best trade-off between area and performance for the current mapping of the NoC and buses.

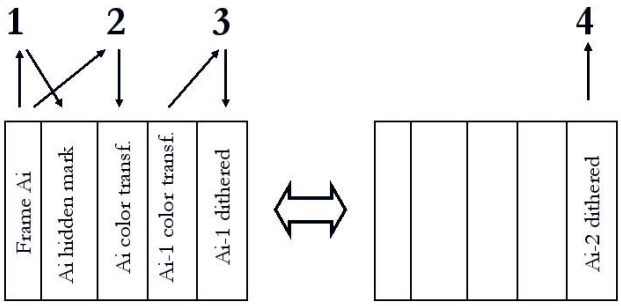


Figure 4. Test application 2: Multiprocessor image pipeline

Table 3
Overall results obtained in Case Study 2

	Bus	NoC, prefetching (8 words)	NoC, no prefetching
T (cycles)	2.927×10^{6}	1.708×10^{6}	3.381×10^{6}
BW (bytes/cycle)	0.113	0.193	0.097
BW/slices	3.7×10^{-5}	2.03×10^{-5}	1.02×10^{-5}

5.2. Multiprocessor image pipeline

In the second test a parallel application implements an image pipeline in four processors. Three processing elements process image data in one memory and the remaining CPU extracts in parallel results from the other memory and loads the new data. Then, the roles are exchanged for the next frame chaining the process (see Figure 4). Hardware semaphores are used to synchronise the processes.

A significant improvement of more than 41% in speed is attained using the NoC with prefetching (see Table 3) and, as a result, the efficiency coefficient of the implementation using the NoC with buffer is increased. However, it still continues being lower than for the bus ($3.7\times10^{\cdot5}$ vs 2.03×10^{-5}). Again, the main conclusion is that the application requirements determine the best option for the developer. However, with this topology the bus clearly achieves a better use of the FPGA resources as far as transmission efficiency is concerned.

6. Conclussions and Future Work

In this paper we have presented a new MPSoC platform that enables the execution of complex real-life applications and to study different interconnection alternatives. It is implemented in state-of-the-art FPGA technology. The system is easily scalable and the proposed architecture can support any number of IPs with the only limit of HW availability. Even systems that cannot be implemented in real-life can be used for architectural experiments through RTL simulation. Then, we have illustrated its versatility with an example of customization of the architecture with four general-purpose processing cores, and implemented with a high-performance bus and a state-of-the-art NoC. We have also provided typical parallel graphic applications and analyzed the performance of each type of interconnection. Finally, as an example of the utility of the proposed platform, we have studied the possible trade-offs between performance and area for both bus and NoC-based designs.

Our results suggest a number of possible future research lines. First, further work needs to be carried out to scale the platform to a larger variety of IPs. More specifically, it would be interesting

to add specific DSP-like cores and more types of existing memories, and we plan to incorpore the hard-wired PowerPC cores available on the used Xilinx FPGA for our platform. Second, because software and system nature determines the most convenient interconnection, we want to investigate this relationship more in deep, namely, we would like to know where a bus can be used without complexity or performance problems and, conversely, when a NoC is more appropriate. Third, our experiments indicate that the behaviour of the NoC can be significantly improved if simple tailor-made optimizations are added according to the eventual type of running applications. For instance, NoC performance has been remarkably enhanced by a simple prefetch buffer. Moreover, instead of using this simple mechanism, it can be considered the use of real data caches to obtain additional performance gains. Finally, due to the synthesis results obtained with the NoC, we consider that there is a big room for possible improvements by developing a network without abstracting the physical device's architecture. In this context, the design of switches according to the specific architecture of FPGAs could clearly achieve extensive benefits.

Acknowledgements

This work is partially supported by the Spanish Government Research Grant TIC2002/0750 and a Mobility Post-doc Grant from UCM for David Atienza.

References

[1] Luca Benini and Giovanni De Micheli. Networks on chip: a new soc paradigm. *IEEE Computer*, 35(1):70–78, January 2002.
[2] Gordon Brebner et al. Networking on chip with platform fpgas. In *Proc. of FPT*, 2003.
[3] Doris Ching et al. Integrated modeling and generation of a reconfigurable network-on-chip. In *Proc. of IPDPS*, 2004.
[4] Nicolas Genko, et al. A complete network-on-chip emulation framework. In *Proc. of DATE)*, 2005.
[5] Antoine Jalabert et al. xpipescompiler: A tool for instantiating application specific networks on chip. In *Proc. of DATE*, 2004.
[6] S. Kolson et al. A network on chip architecture and design methodology. In *Proc. of ISVLSI*, 2002.
[7] Theodore Marescaux et al. Networks on chip as hardware components of an os for reconfigurable systems. In *Proc. of FPL*, 2003.
[8] Fernando Moraes et al. Hermes: an infrastructure for low area overhead packet-switching networks on chip. *Integration, the VLSI Journal*, April 2004.
[9] OCP International Partnership (OCP-IP). Open core protocol standard, 2003. http://www.ocpip.org/home.
[10] David Siguenza-Tortosa et al. Vhdl-based simulation environment for proteo noc. In *Proc. of HLDVT Workshop*, 2002.
[11] C.A. Zeferino, et al. Rasoc: a router soft-core for networks-on-chip. In *Proc. of DATE*, 2004.
[12] A. Jerraya and W. Wolf. *Multiprocessor Systems-on-Chips*. Morgan Kaufmann, Elsevier, 2005.
[13] C.A. Zeferino et al. Socin: a parametric and scalable network-on-chip. In *Proc. of SBCCI*, 2003.
[14] Davide Bertozzi and Luca Benini. Xpipes: A Network-on-Chip Architecture for Gigascale Systems-on-Chip. In *IEEE Circuits and Systems Magazine*, 2004.
[15] Kees Goossens, et al. The Aethereal network on chip: Concepts, architectures and implementations. In *IEEE Design and Test of Computers*, Vol 22(5):21–31, Sept-Oct 2005.
[16] Xilinx Inc. The Virtex2 Pro HandBook.
[17] Xilinx Inc. MicroBlaze Processor User-Guide.
[18] Synopsys Inc. The Synopsys Design Compiler User Guide.
[19] Xilinx Inc. On-Chip Peripheral Bus v2.0 with OPB Arbiter.

A New Model for NoC-based Distributed Heterogeneous System Design*

F. Rincón[a], F. Moya[a], J. Barba[a], D. Villa[a], F. J. Villanueva[a] and J. C. López[a]

[a] School of Computer Science, Universidad de Castilla-La Mancha, Paseo de la Universidad 4, 13071 Ciudad Real, Spain

This work explores the capabilities of a new design methodology aimed at offering an integrated and homogeneous way of thinking on the whole system. This methodology allows extending a widespread communication model, the remote method invocation model, to the design of a NoC system as a part of a more complex distributed system.

Some standardization efforts, such as OCP, have tried to define a common syntax for communications among NoC components but there is not a provision for a common semantics yet. On the other hand, there is a dramatic variation of the communication capabilities between two cores depending on the relative location of the two peers (on- and off- chip). Since a common communications infrastructure is missing, on-chip functionality may only be accessed from off-chip components using ad-hoc interfaces that exists only if it has been foreseen by the designer. In this work we will discuss the way our proposed methodology is able to tackle these and many other issues without major overhead.

1. Introduction

The Network on Chip (NoC) design paradigm, as an enabling technology for the integration of a high number of computational blocks interacting with each other, provides the adequate way to face an important part of the design of these applications. Taking the network as a facilitator to overcome complexity and scalability, NoCs face the main problems that arise when designing complex SoCs (System on Chip) composed of dozens, maybe hundreds, of IPs communicating with each other.

But these NoCs must interact with many other systems implemented in a vast range of technologies and using again the network as the basis to reach the goals of the new age applications: ubiquitous computing, cooperation between applications, knowledge management, multimedia communication, intelligent and sustainable growth… Besides the heterogeneity of the different system components, the resulting systems use also different and heterogeneous means of communication. A new concept appears, the distributed heterogeneous system, where the components are defined by the service they offer to the rest of the system, independently of their implementation and their location. This way of thinking on the functionality of a system poses new design challenges that claim for new design methodologies able to face them, keeping always in mind the reduction of the design time.

This work explores the capabilities of a new design methodology based on these concepts and able to offer an integrated and homogeneous way of thinking on the whole system. This methodology allows extending the remote method invocation model, to the design of a NoC system as a part of a more complex distributed system. Some standardization efforts, such as OCP[1], have tried to overcome the interface compatibility problem defining a common syntax for communications. But there is not a provision for a common semantics yet. On the other hand, there is a dramatic variation of the communication capabilities between two cores depending on the relative location of the two peers (on- and off- chip). Since a common communications infrastructure is missing, on-chip functionality may only be accessed from off-

* This work was supported by the Spanish Ministry of Education under Grants TIN2005-08719, and TEC2004-05205, and by Junta de Comunidades de Castilla-La Mancha under Grants PBI-05-049, and PBC-05-009.

chip components using an ad-hoc interface that exists only if it has been foreseen by the designer.

In this paper we will discuss the way our proposed methodology is able to tackle these and many other issues. We will first describe the remote method invocation model and how it is applied to NoCs, leading to a better separation of concerns between system design and system deployment. A thorough description of the mapping of this model onto an OCP based architecture will be done. We will them go through the different benefits and other consequences of applying such as flexible and homogeneous view of the system, including an analysis of the potential overhead.

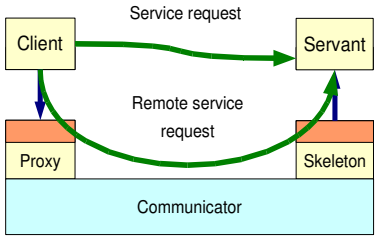


Figure 1. RMI provides an abstracted view of the communications channels.

2. Remote method invocation

Remote method invocation (RMI) is a synchronous communication model already popular in object-oriented distributed software systems [2][3][4]. RMI (see Figure 1) tries to emulate the behavior of a normal method invocation and dispatch of object-oriented programming languages. Objects are passive entities incarnated by some particular implementation (servant) waiting for requests to arrive somewhere in the network. A client requests a service from an object by issuing a RMI and blocks until the object replies. This is mostly the same semantics of the usual method invocation.

A common missconception is that such a synchronous model prevents parallel computations. This is not even the case of distributed software implementations, since both, client call and object dispatch may be handled by a different thread.

Figure 1 shows a simplified view of the logical structure. We may identify two roles for communicating objects. Servants (object implementations) are passive entities which share a set of services through a specific interface. On the other hand, clients may initiate a request following the interface exposed by the servant. A third component, the communications engine, act as a mediator between client and servant. Servant location, message routing, and network bridging are all responsibilities of the communications engine, or just communicator for short.

Actual connection of clients and objects with the communicator is made through entities that may be automatically generated from abstract object interface descriptions. In the client side there are *proxies*, which provide the illusion of always using local servants while they may actually reside on another computing device. On the object side there is a *skeleton*, which is resposible for adaptation of the specific servant interface and the messages in the network.

Objects and object implementations (servants) are kept as different concepts in order to easily provide many advanced features (object persistence, transparent replication, implicit activation, resource sharing, etc.). Therefore an object is an entirely abstract concept which is mapped to a specific servant by the communicator at method invocation time. Many popular software architectures fit in this model, such as CORBA[3], Java RMI[2], .NET Remoting[4], EJB[5], ICE[6], etc.

An independent communications engine component makes easier to transparently add a whole set of advanced features. One of the most relevant for this work is the ability to indirectly reference objects in order to achieve complete location transparency. That is, a method invocation may be issued to an object whose location is determined at run-time. Some important applications of this feature are the upgrading of on-chip components using off-chip modules, object migration to other locations, implicit object activation, automatic load balancing, etc.

The communicator keeps a mapping of objects and implementations (usually called *servants*)

through another component called *Object Adapter*. There is no need for a one-to-one mapping. It is possible to *incarnate* an object in more than one servants (e.g. fault tolerance through replication), and we may also reduce resource consumption by implementing a set of objects with a single servant. It is also possible to have objects which are not incarnated in any servant at a given time, and implicitly instantiate a servant at invocation time. We see this feature as a key factor for easier development of dynamically re-configurable systems.

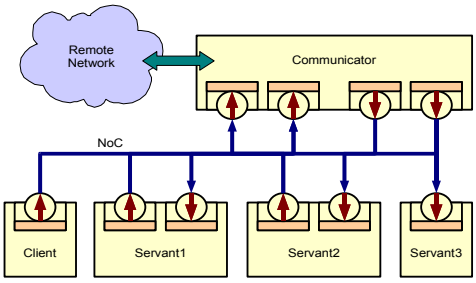


Figure 2: Simplified view of the middleware on a NoC.

The communicator effectively hides network details from clients and object implementations. Communicators on different platforms must agree on a shared protocol in order to allow remote interoperability, but components inside a single platform may optimize their internal communications, as we will see in the following sections. Each major distributed software middleware define a specific inter-communicator protocol (e.g. GIOP in CORBA and EJB, SOAP in .NET Remoting or WebServices, IceP in Internet Communications Engine, etc.).

These few simple abstractions provide a complete object-oriented framework for hardware modeling. Servants encapsulate functionality and state incarnating objects when they are needed. Proxies behave as low-cost references to remote objects, and skeletons translate network messages into local service requests. But designers of a NoC based system should also care about the overhead of this approach. A pure hardware design is much more static than a conventional software system and therefore it is hard to justify a significant overall overhead for rarely used dynamic object management features. In the following sections we will discuss in detail the implementation of the proposed middleware with minimum overhead as one of the major design constraints.

3. Global Communications System

The model described in section 2 does not depend on a particular communication protocol or data transport layer. Nonetheless we will describe a proposed mapping of this model to an OCP based architecture. The hardware implementation of the communications engine follows a logical structure similar to what is shown in Figure 2. Hardware objects are implemented by servants registered in an object adapter inside the communicator. The object adapter receives messages from the remote network and forwards them to the right servant. From a hardware point of view the communicator is a core which includes a number of general purpose services such as object adapters, remote servants, indirect servants, and communication primitives for inter-communicator interoperability. We will discuss the implementation of these features in the following subsections.

3.1 Hardware Objects

The first problem to be addressed is the physical implementation of objects. Traditional objects are dynamic entities, that may be created and destroyed in run-time. Their methods should be dispatched at run-time, rather than using static connections, in order to be able to implement inheritance and polimorphism. On the other hand, hardware is fundamentally static with the exception of dynamically reconfigurable systems.

An implementation of the object abstraction as it is known in the software domain would lead to unacceptable overheads due to the complex management mechanisms required. We limit the hardware support for objects to the bare minimum features which are obviously useful in the hardware domain and do not impose excesive overheads. We focus on dynamic creation and

destruction, separation of objects and implementations in order to maximize reuse, and low overhead object references. We explicitly avoid dynamic dispatching and run-time operation binding to avoid excesive overheads.

A hardware object provides just an encapsulation mechanism for some functionality (methods), shared among all of the instances of each class, and some per-instance private state data (attributes). Besides, methods are always considered static while attributes may change along the object lifetime. The semantics we use for objects is quite compatible to the traditional implementation of distributed objects (CORBA, EJB, .NET Remoting, etc.). The logical structure of a servant is shown in Figure 5 and will be described later.

The communicator provides an addressing mechanism to redirect object method invocations to servants. Each servant contains the physical implementation of all methods defined in a class and it is also responsible for managing per-object state data (attributes). This can be achieved with some storage elements (a local memory, a shared memory, a register file, etc.). Our proposed model do not impose a particular state storage method. Sometimes a shared memory will be used for state data storage. Other applications will require persistent storage in a FLASH memory. Sometimes we will instantiate a servant to implement a single object and therefore there is no need to keep several object states. It is even possible to define state-less objects. Since there is no way to directly access the state data from the clients we can be sure that the alternative chosen by the servant designer will not affect the rest of the architecture. Sharing servants among a set of objects is an easy way to reuse design components. If properly designed these shared servants do not prevent parallel execution of methods (e.g. by using a pipeline).

Each method invocation is characterized by a unique destination object identifier (*obj_id*), a unique operation identifier (method) and the set of parameters for that operation. Servants are quite similar to current IP blocks. Therefore legacy IP blocks may easily be integrated in our middleware based architecture incarnating a single object.

3.2 Proxies and skeletons

Interoperability is an additional requirement for servants not so commonly found in commercial hardware components. This is achieved by means of standardized interfaces called proxies and skeletons. Any method invocation must take place between a proxy and a skeleton. From the client point of view a proxy is seen as a private object implementation. It provides exactly the same physical interface. Servants on the other hand do not need to care about the location of clients. They just provide an object interface and export that interface through a skeleton.

For example, assume that the hardware middleware uses OCP as the communication protocol for some objects. Clients and servants are just cores that must behave as masters and slaves of the OCP bus. Therefore we need to translate the object interface (operations, parameters, ...) into OCP signals. On the client side proxies generate OCP messages from the invocations received. Skeletons receive OCP messages and translate them into signals of the object interface.

Clients and servants are not aware of the existence of an OCP bus. Indeed there is no need to use OCP. We propose OCP because it is a low overhead standard, but the communication protocol is abstracted by proxies and clients. This is an important observation for the designer since proxies and clients may be generated automatically from an abstract description of the object interfaces. Most modern distributed software middlewares already provide an interface description language for that purpose. Translating such interface descriptions into hardware requires the precise definition of a set of technology mapping rules giving a corresponding physical interface for each abstract object interface. For example, an operation may be mapped

into a one-bit input port. Designers and proxy/skeleton generators must adhere to that mapping rules.

Proxy and skeleton overhead may be negligible in most cases. A designer using OCP must already define an adapter to guarantee logical compatibility. Proxies and skeletons are equivalent to those adapters from the remote invocation point of view.

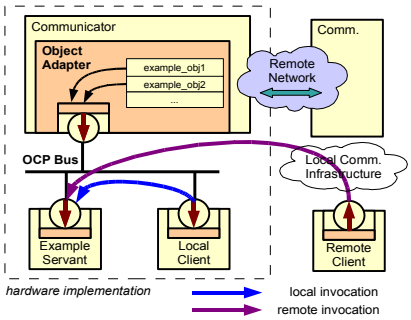


Figure 3: An object adapter makes on-chip objects available to off-chip components.

3.3 Object adapter

When the communications engine receives a remote invocation (from an off-chip component) it must first determine which servant should handle it. That is the role of a communicator component called the *object adapter*.

The incoming message contains an identifier of the destination object. This identifier is used to index a table (*Active Object Map*, see Figure 3) which relates each object identifier with the corresponding servant. Any object whose identifier is not registered in the active object map cannot be accessed from the remote network. On the other side, servants not incarnating any object are considered inactive, but they may still be used locally.

Communication between object adapter and servants is performed as described in section 3.2. The object adapter holds a collection of proxies used to contact the destination servants. As stated before, several objects may be incarnated by a single servant (entries in the *Active Object Map* pointing to the same servant), or a single object may be replicated on several servants (replicated entries in the *Active Object Map*). Note that the middleware does not guarantee replication transparency on the servant side. Servants are responsible for consistency management.

Efficiency and low overhead are two major concerns in a hardware communications framework. The model described allows run-time registration of new servants in the *Object Adapter* and dynamic control of remote accessibility of new objects by adding new entries in the *Active Object Map*. Both are feasible operations in hardware. Registering new servants may require adding new proxies to the object adapter if they are not known in advance. That feature may be provided by dynamically reconfigurable devices.

Most of the communications infrastructure is either generated automatically (skeletons and proxies) or provided by the hardware communicator. The system designer may then postpone decisions about the system deployment until the latest stages of the design flow.

3.4 Local Objects and Remote Objects

As described above proxies and skeletons provide network transparency. In most distributed software middlewares proxies are opaque entities which encapsulate local or remote access to the destination object. Local accesses may be easily optimized by direct translation of remote invocations to standard method calls. These are usually called *co-located servants*. Our proposed architecture also allows direct local object interaction, without the help of the communications engine.

Hardware proxies are not required to store any information on the physical location of remote objects. The communicator provides special purpose remote servants which translate invocations to remote objects into messages at the remote network (see Figure 4). These *remote servants* may also be automatically

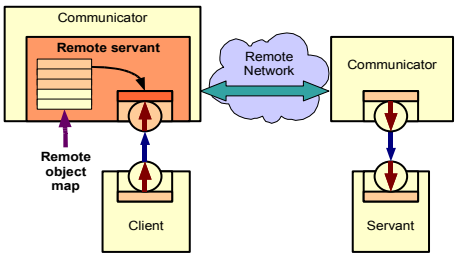


Figure 4: From the point of view of the proxy there is no difference between local and remote objects.

generated.

We consider local invocations by default handling the remote objects as a special case. Software middlewares do exactly the opposite. Each remote servant holds a *Remote Object Map* which relates local bus addresses with global object identifiers as used in the remote network. In contrast to software middlewares our approach introduces almost zero overhead for local communications, which is a major concern for hardware designs.

A middleware with the previously described mechanisms already allows a large degree of location transparency. The design of the different entities in the system is independent of the physical location of each one. It is even possible to implement some entities in software without modifications to the other entities. But modern distributed software middlewares allow run-time location of objects. This is specially interesting for hardware designs because it would allow component upgrading even after fabrication.

We provide this level of location transparency through another communicator component, an specialized *indirect servant*. Proxy to servant protocol must be augmented to include a *redirect* message. At invocation time the indirect servant examines where is actually located the destination object and emits a *redirect* message containing the actual address of the destination object. When a proxy receives a *redirect* message it must retry the invocation using the received data. This kind of proxies are a bit more complex but they are only needed when a higher level of location transparency is required. Given that both proxies and skeletons are automatically generated, the designer may evaluate the trade-off between flexibility and overhead to suit his needs.

4. Experimental results

Let us illustrate the discussion in the previous sections with a little example. Figure 5 shows the UML definition of the example_class from which some objects will be derived. The class only contains the value attribute and two methods to read and write the attribute. This extreme simplicity has been choosen for the sake of clarity, since the concepts presented here could be generalized to more real and hence much more complex situations.

We will assume that the system architect already decided to make hardware implementation of the objects of this class remotely accessible from anywhere in the network. So the implementation will consider two different situations for the invocation of the methods: local and remote communication.

As already mentioned in the description of the comunication engine, that will not always be the case for all hardware objects, since many of them will only be useful for local computations. In that case the servant implementing the objects will only be accesible through the local communication infrastructre. Besides, we will also assume the use of OCP to provide the transport protocol between local comunicating entities. Remote comunication can be provided through any kind of network interface. Figure 5 shows the structure of the example servant. We can distinguish two main parts, one related to the implementation of the objects of the example class, and the skeleton that will provide access to the methods of the class. Objects are accessed through a component whose interface mimics that of the object plus some extra information such as the object identification (*obj_id*). The state of all the objects is stored separately and can be addressed through the object indentification. The second part of the servant is the skeleton that translates the local comunication protocol to an operation invocation. One important thing to note is that skeletons are automatically generated from the interface definition of the object (once the mapping rules for the local protocol have been defined). Since local comunications are performed over an OCP logic bus, the skeleton is in fact an OCP to the object interface slave bridge.

For a client to perform any invocation, a proxy with the target object interface must be

implemented. The proxy mirrors the skeleton, providing the inverse, object interface to OCP, translation (see Figure 6). The proxy is also generated automatically from the same object interface description than the skeleton, so it can be easily optimised for the application. It could be possible, for example, to include in the proxy only those operations that will be invoked by the client.

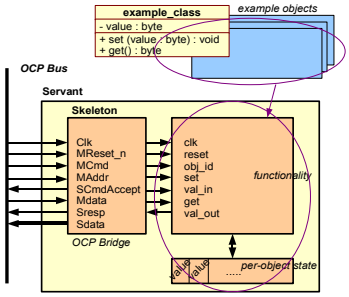


Figure 5: Simple servant handling multiple instances.

As described in section 3 remote invocations will be handled by a specialized communicator component named the *object adapter*. The object adapter is composed of two parts. 1) On one side it holds a finite state machine for message handling (parsing and building), and for marshalling and unmarshalling of the available data types in the network. The precise definition of this state machine depends on the selected middleware protocol (GIOP for CORBA interoperability, IceP for ICE interoperability, etc.) but there is no need to re-design it for each application. 2) After successful reception of a message, the object adapter locates the destination object by means of a lookup table and routes the raw data through the corresponding proxy.

The process for a local invocation is the following. 1) The client activates the corresponding signals of the local proxy component. 2) The proxy translates the invocation into a an OCP master comand that establishes a point to point comunication between the client and the servant (whose address is coded in the proxy). 3) The OCP slave (the skeleton) receives the command through the NoC that is translated into the proper signal activations of the objects component interface. 4) The operation is executed using the corresponding state information.

When an on-chip component invokes a method of an off-chip object then a remote servant comes into play. The remote servant redirects the incoming data through the network using the appropriate marshalling and protocols. It is obvious that the remote servants share a lot of capabilities with object adapters and therefore middleware implementors may share communicator resources between these two components.

As shown in this example, proxies and skeletons are mapped to simple OCP adapters. Therefore the system will even work in the absence of a communicator. In any case, the system designer may evaluate at any point of the design flow the tradeoff between minimum cost implementations and flexibility of the deployed system.

5. Implications of a Hardware Middleware

As described above, a hardware middleware introduce a relatively small set of concepts and the physical implementation is straightforward. One may be tempted to think that the architecture we propose is just a minor evolution of traditional hardware design and miss the major strenghts of our approach.

From the point of view of the design methodology, a fully functional simulator of the whole system may be developed in the early stages of the design on a single computer using a standard middleware. Afterwards, in every step of the design cycle there is a complete functional system. Simulated software components will directly interact with a partial NoC design without any special consideration, leading to easier system debugging. System deployment and global system partitioning will constitute a new orthogonal role.

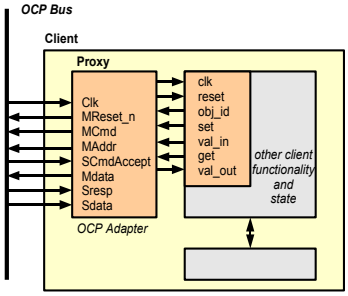


Figure 6: Proxy using an OCP master to interface the NoC.

There has been an intense debate on which place should the operating system occupy in system design. Proposals range

from a single HW/SW operating system for the whole system to the concept of distributed nanokernels. Fortunately the communications engine is independent of the operating system. If required, it may even be implemented on top of the hardware-software middleware leveraging the transparency features of the communicator.

From the point of view of the customer, a hardware middleware allows dynamic on-line upgrades of systems without disturbing their normal operation. A new implementation of a given module may be added to the system by just adding an entry in an indirect servant.

A hardware middleware opens a whole new range of possibilities we did not consider in this paper due to space constraints. For example, it is feasible to automatically generate pieces of hardware to automate state storage and recovery. This would be the first step in the development of transparent hardware object migration, specially interesting for dynamically reconfigurable distributed systems.

The introduced overhead depends on the final deployment of the whole system. For example, many hardware objects should not be accessed from the remote network. In this case there is a minimum overhead since we use the same logical buses or NoCs as middleware-less designs.

Efficiency considerations may lead to even deploy point-to-point communications between some pairs of clients and servants. Besides, NoCs used for proxy to servant communications may be optimized as usual. For example, it is possible use a segmented bus, or a complete switched NoC with OCP interfaces. This is entirely orthogonal to our approach.

6. Conclusion

The middleware based methodology proposed in this paper introduces very few abstractions and may easily be integrated in current design flows. In spite of its simplicity it provides a vast range of benefits which are briefly summarized below.

1) It provides a low-cost hardware object-oriented framework which is quite handful to integrate system-level object oriented modeling such as UML.

2) It provides a high degree of transparency for system designers. NoC transparency is achieved with proxies and skeletons, remote network transparency is provided by the *remote servant* and the *object adapter*. Full location transparency is provided by the *indirect servant*.

3) It also provides some advanced features such as basic support for load balancing, fault tolerance through replication, and transparent instantiation of dynamically reconfigurable hardware. These mechanisms set up the basis for object persistence and object migration.

We believe all these features cooperate nicely to provide better orthogonalization of concerns and an integrated and homogeneous view of the whole NoC based system. The introduced overhead may easily be tuned at late stages of the design flow.

7. References

[1] *Open Core Protocol Specification, Release 2.0*, OCP-IP Association 2003.
[2] *Java™ Remote Method Invocation Specification*, Sun Microsystems, Inc. 2003.
[3] *The Common Object Request Broker: Architecture and Specification, Revision 2.6.1*, Object Management Group 2002.
[4] Piet Obermeyer, and Jonathan Hawkins. (2001, July). *Microsoft .NET Remoting: A Technical Overview* [Online]. Available: http://msdn.microsoft.com/ library/en-us/dndotnet/html/hawkremoting.asp
[5] *Enterprise JavaBeans Specification, Version 2.1*, Sun Microsystems 2003.
[6] Michi Henning, and Mark Spruiell, *Distributed Programming with Ice*, Palm Beach Gardens, FL: ZeroC, Inc. 2005.

Minisymposium

Skeletons

Integrating MPI-Skeletons with Web Services for Grid Programming

Jan Dünnweber[a], Anne Benoit[b], Murray Cole[b], Sergei Gorlatch[a]

[a]University of Münster, Münster, Germany

[b]School of Informatics, The University of Edinburgh, Scotland, UK

Interoperating components, implemented in multiple programming languages, are one of the key requirements of grid computing that operates over the borders of individual hardware and software platforms. Modern grid middleware like WSRF facilitates interoperability through service-orientation, but it also increases application complexity. We show that Higher-Order Components (HOCs) provide a service-oriented programming abstraction over middleware technology. By offering a MPI-based skeleton implementation as a HOC, we show how machine-oriented technologies can be integrated with grid middleware and made available via Web Services. The interoperability features of this HOC are demonstrated by binding it to a Java-based Web application, which allows to transform user defined input in a highly effective manner by running wavelet computations remotely on parallel machines.

1. Introduction

A grid infrastructure connects computers of varying architectures, so that any task in an application can be delegated to the most appropriate processing platform. Programmers targeting a grid currently face a tradeoff when choosing the implementation technology for their applications. Machine-oriented technologies [5] provide good performance, but they narrow the range of the possible execution platforms. This is due to the fact that C is compiled into native machine code, which cannot be interchanged among different machines offering unequal instruction sets. Moreover, the use of function pointers, as it is required, e. g. , for MPI collective operations implies an undesirable tight coupling between software components: code of library functions implementing a generic functionality must be present in the same address space as application-specific parameter code.

In contrast, a service-oriented architecture (SOA [4]) based on grid middleware such as WSRF [9] loosely interconnects clients and compute nodes, i. e. , they may be interchanged without affecting the application code. The communication is handled via Web Service requests and the required APIs for issuing and processing such requests are available for interpreted languages and also for C. Despite of the gained interoperability advantages, the use of Web Services for handling the entire communication in an application usually imposes a loss of performance. The messaging protocol employed by Web Services is SOAP, which requires the time-consuming composition, transmission and parsing of an XML-tree structure, even for elementary data exchange.

Our goal is to combine the high-performance approach to parallel programming using C and MPI with the recent SOA efforts in grid computing via the provision of Higher-Order Components (HOCs), which can abstract over the technical details of communication on the grid as explained in [6]. In this paper, we introduce a gateway service bridging between MPI and SOAP, which allows to use MPI inside a HOC implementation.

The next section shows how HOCs can be integrated with MPI for incorporating code based on traditional messaging primitives with recent middleware. Then, in Section 3 we introduce the case study of the discrete wavelet transform (*dwt*). Section 4 presents an imaging application using our MPI-based HOC. We explain our gateway service in Section 5 and conclude in Section 6.

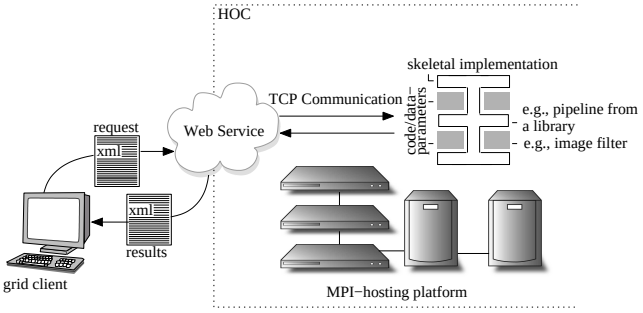


Figure 1. Abstracting over the runtime platform using a Higher-Order Component (HOC)

2. Integrating HOCs with C and MPI

Higher-Order Components (HOCs) were introduced in [6] to abstract over grid middleware and make the required middleware setup transparent for the Java-programmer. Of course, the same construct is desirable for C and MPI programmers as well.

HOCs offer a skeleton-like programming interface: the HOC corresponding to the map skeleton, e. g. , provides a service that applies functions in parallel to parts of its input. Its code parameter is the mapped function, which is portably represented by a string in the HOC implementation. HOCs and their parameters correlate to the skeleton model [2], but the implementation of a HOC takes into account the distinctive features of a SOA, e. g. , there is no standard format defined to exchange executable code between Web Services. HOCs include a grid-aware mechanism for shipping units of executable code across the network.

The most notable difference between a HOC and Web Service based job submission system, such as the Globus resource allocation manager (WS GRAM) or Unicore/WS [14], is that, in case of using a HOC, a skeletal implementation of a parallel algorithm is deployed upon the runtime platform before the HOC is used in an application. Figure 1 schematically depicts this scenario: Instead of the complete application code, only application specific code and data parameters are uploaded, when XML-encoded service requests are sent by the client. for the given application. The parallel implementation for processing the request remotely in the grid can make use of MPI as suggested in the figure, or it can comprise multiple interconnected Web Services providing an alternative parallel processing platform, as described in [6].

The HOC, which we integrate with MPI makes use of the C-library *eSkel* [3]. *eSkel* provides us with ready-made implementations of reusable patterns for parallel processing in the form of MPI-skeletons, i. e. , the code for running general schemata on top of MPI is given, but application specific code must be provided to the skeleton via parameters. Contrary to a typical *eSkel*-application, where the client is running on top of the MPI platform itself as, e. g. process 0, the HOC client connects to a Web Service which maintains a TCP-connection to one process dedicated for handling the external communication (Section 5 explains this gateway in more detail). Our HOC not only abstracts over the skeleton implementation, but it also decouples the client from the skeleton, thus allowing both implementations to be exchanged without affecting each other's code and promoting code reusability.

Our application used for a case-study in this paper has a pipeline structure: the discrete wavelet transform, which can be decomposed into multiple successive steps. Implementing this transform by mapping stages of the underlying pipeline model to different processors leads to frequent inter-processor communications, because of the fine grain of the single-stage operations. Therefore, a

parallelisation should employ a light-weight message passing mechanism. Therefore, the MPI-based pipeline skeleton offered by *eSkel* is a suitable candidate.

By embedding *eSkel*'s pipeline into a Web Service that offers it as a HOC, we provide an interface to any Internet client allowing to access it remotely via SOAP. The new HOC accepts parameter functions, which are shipped over the network and may be sent from a service consumer; the latter may be implemented in a programming language other than C. As an experiment, we connected the pipeline service to a Java-based Web interface which allows the user to upload and transform data via a Web browser.

3. Case Study: The Discrete Wavelet Transform

Wavelet transform is used in applications such as equalising measurements, denoising graphics and data compression. Such procedures are often applied iteratively to large amounts of data, which is time-consuming. Therefore a grid-enabled implementation which allows for the outsourcing of computations to high-performance multiprocessor servers is desirable. In an application, the transform is customised for a particular objective so that the transformed data exhibits properties which cannot be detected so easily in the source data. As an example, the contours in an image can be accentuated. Another popular application of wavelets is data compression via a customisation of the transform where the resultant data can be represented using less memory. Customising the wavelet transform is done by parameterising a general schema with application specific functions.

3.1. The Wavelet Lifting Scheme

Wavelet transforms are integral transforms, closely related to the (windowed) fast Fourier transform (*fft*), which decomposes a function into sines and cosines. The continuous wavelet transform (*cwt*) defined below projects function $f(t)$ onto a family of zero-mean functions (*wavelets* ψ):

$$cwt(f;a,b) := \int_{-\infty}^{\infty} f(t)a^{-\frac{1}{2}}\overline{\psi}(a^{-1}(t-b))\mathrm{d}t \tag{1}$$

Instead of a continuous function, the discrete wavelet transform (*dwt*) processes a set x of samples (such as a list or a matrix) which are subdivided into two equally sized, discrete subsets u and v. The "lifting technique", proposed by W. Sweldens in 1994 [12], allows us to compute *dwt* iteratively as follows. Initially, $u_{0,j} = x_j$ for $j = 0..m_0$. The first index of $u_{i,j}$ (index i) represents the *lifting step*, and m_0 is the initial number of elements. $dwt(x)$ is computed by applying two functions called *predict* and *update* repeatedly, according to the following *lifting scheme*:

$$\begin{array}{ll} (u_i, v_i) := split(u_{i-1}) & \\ u_{i+1,j} := u_{i,j} - predict(v_{i,j}) & \text{for j} < \mathrm{m_i} \\ v_{i+1,j-m_i} := v_{i,j-m_i} + update(u_{i+1,j-m_i}) & \text{for j} \geq \mathrm{m_i} \end{array} \tag{2}$$

At each increment of i, index j iterates from 0 to $2m_i$ to complete one *lifting step* (first step corresponds to $i = 1$). First, the set u_{i-1} (of size m_{i-1}) is split into subsets u_i and v_i, which are thus of size m_i ($m_i = m_{i-1}/2$). The *predict* function is then applied to the values in subset v_i ("predicting" the values in subset u_i). The samples u_i are then replaced by the differences between their predicted values and their actual values. These differences are processed by the *update* function and added to the samples in subset v_i ("correcting" it). Note that the workspace, i. e. , the data that is affected by subsequent steps is reduced in each lifting step: once computed, all $v_{i,j}$ values remain unchanged.

(a) wavelet analysis (b) wavelet synthesis

Figure 2. The lifting scheme

The wiring diagram of two lifting steps in Figure 2(a) illustrates the structure of the lifting algorithm. The minus indicates that the input from the top is subtracted from the input from the left.

While the computation schema (2) is fixed, the functions *split*, *predict* and *update* can be customised. This customisation is done by the user, depending on the characteristics of the application. For example, if plain number series are processed, the *split* function can simply be defined to separate entries with odd and even indices. The choice of suitable *update* and *predict* functions requires making an appropriate assumption about the correlation of the single elements within the processed data.

3.2. Parallelising the Lifting Scheme

When the lifting scheme is applied to multiple independent data sets in parallel, the pipeline skeleton [2] can be used to parallelise the computation. The number of lifting steps that we apply to an input set (called the *scale* of the transform in classical wavelet theory [7]) depends on the size of the set (m_0): the number of steps is $log_2(m_0)$, since the input is bisected at each step. For a straightforward parallelisation, we use a pipeline wherein each stage corresponds to one lifting step and the number of stages is determined by the largest input set.

Wavelet transformation is reversible: the original input can be reconstructed from the transformed data using an inverse process, called the wavelet synthesis. In this context, the forward transform is usually called wavelet analysis. Our pipeline-based implementation of the wavelet transform allows us to run both a wavelet analysis and a wavelet synthesis. In the reversed schema, update and predict functions are swapped; updated values are subtracted and predicted ones are added as shown in Figure 2(b). A reverse pipeline with the same number of stages as the wavelet analysis pipeline can be used to reconstruct the source data. We use this output data for a comparison with the original input to verify the correctness of our implementation. The reverse process has another notable property: the workspace increases, since it re-introduces the stored v_{i+1} values to compute u_i (see Section 3.1).

3.3. An Application to Image Data

Figure 3 shows the effect of an example application of *dwt* on images. The input image is transformed up to the maximum scale and then reconstructed via the inverse transform as introduced in Section 3.2. The fractal image in Figure 3(a) is a Julia Set for $c = -0.16 - 0.65i$ (to construct such diagrams, see [11]). It features very fine contours that become bolder in the reconstruction (Figure 3(b)), since all the pixel values below a given threshold are set to zero in the transform.

Contrary to the elementary splitting of number series, explained in Section 3.1, images require the splitting to be customised via a parameter function specifying a 2-dimensional partitioning. If we simply concatenate all rows or all columns of the image matrix into a 1-dimensional array, the image structure would be lost during the transform, as most neighbouring entries in the matrix are

disjoint in such an array. Instead, we overlay the image with a lattice that classifies the pixels into two complementary partitions, preserving the data correlations.

4. Running the Pipeline HOC, customised for the Lifting Scheme

The middleware setup for running Web Service-based applications can be intricate. This is especially the case in the context of WSRF or Unicore/WS, where Web Service operations can affect component states, represented via resource properties. Moreover Web Service notifications [9] are often used on the grid for asynchronous messaging. Thus, resources and notifications must be configured additionally to Web Services. Each setup step is error-prone and configuration files cannot be debugged by a stepwise execution in the manner of traditional executable code.

For programming the wavelet lifting scheme in parallel, we use a higher-level programming interface, the Pipeline HOC, which allows any pipeline-structured application to run on the grid. In Section 3.2, we have shown how to parallelise the lifting scheme using a pipeline skeleton. We explain here how this pipeline can be offered as a HOC and how it can be customised to run the image transformation presented in Section 3.3.

4.1. The Parameter Functions

In the wavelet lifting algorithm, the stages of the pipeline are defined through the parameter functions *split*, *predict* and *update*. For our imaging application, we define a *split* function which computes the so-called *quincunx* lattice. All the pixels of the processed image are alternately assigned to a subgroup of black pixels or white pixels. This quincunx pattern is just one possible partitioning among others. We refer to [8] for details on the implementation of such partitions.

The *predict* function rates the grayscale value of a pixel by computing the average of its nearest neighbours:

$$predict(x_{i,j}) = \frac{1}{4}(x_{i-1,j} + x_{i,j-1} + x_{i+1,j} + x_{i,j+1})$$

The corresponding *update* function returns half of the average computed by the *predict* function, which reflects the bisection performed by the *split* function in each lifting step. In this way, we preserve the average of the input during lifting, i. e. the grayscale value average over all pixels in both partitions equals half the average over all initial values. Some more sophisticated methods also bind neighbouring values, but with a different calculation rule. In any case, both *predict* and *update* can be represented via arithmetic expressions. These expressions can be encoded in an XML-compliant manner using, e. g. , *XPath*-expressions.

XPath is a language designed to select nodes, specify conditions and generate outputs in XML documents [16]. XML processing APIs, such as *Apache Xalan* [10], evaluate *XPath*-expressions and

(a) an example input image (b) reconstruction (threshold 2.5)

Figure 3. Application of the transform on a grayscale fractal image

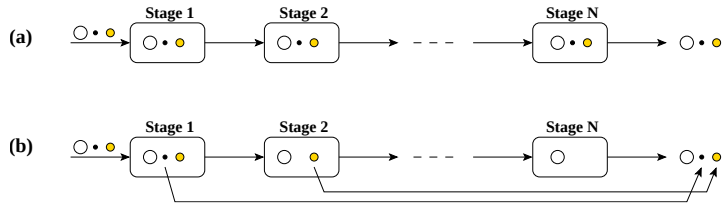


Figure 4. (a) Standard eSkel pipeline skeleton – (b) Stage skipping optimisation

are available for multiple programming languages. We thus consider *XPath* as a suitable format for encoding functions and for exchanging them via the network.

The fractal image computation discussed in [6] uses a loop inside the skeleton parameter function. Loop statements are not supported within *XPath*-expressions and therefore this application requires from the HOC a mechanism to transfer executable code without such limitations.

As traditional HOCs support the transmission of parameters describing complex control flows, such as loops and nesting, in the format of Java or a scripting language, there is still a tradeoff between purely Java-based HOCs and potentially more efficient native skeletons, even if the latter can also be offered as HOCs.

4.2. The Stage-Skipping Optimisation

The HOC can be optimised to shortcut the lifting scheme in order to reduce the number of lifting steps in the pipeline. Indeed, the several inputs of the pipeline may be of varying sizes, and the number of lifting steps is directly related to their size. Hence, an input of short size does not need to go through all the stages of the pipeline. The adaptable design of skeletal componentshelps to address this problem in our implementation.

The parallelisation described in Section 3.2 is not optimal because small-sized sets are to be passed through numerous pipeline stages, although no further processing is necessary. We could therefore envision an optimised pipeline skeleton which skips stages when needed. We perform the computation using the skeleton library *eSkel*, namely its pipeline skeleton which allows us to define so-called *explicit interactions*. In most of the skeletons libraries, the interactions between activities (i.e. the stages of a pipeline) are *implicit*: a pipeline stage is a function which takes input data as a parameter and returns one output for each input, thus being under constraint both in time and space. In *eSkel*, it is possible to define *explicit* interactions [1] between activities and release temporal constraints.

Figure 4(a) displays the standard pipeline behaviour with implicit interactions. The circles represent three examples of input items and the size of the circles is proportional to the size of the input. In our application the inputs of small size do not need to be processed by all the stages. Figure 4(b) presents the stage skipping optimisation, which allows small inputs to skip the unnecessary stages and to be directly sent as an output. As can be seen in Figure 4(b), the smallest input is finished in stage 1 and no more present in stage 2, while it is towed through all stages in Figure 4(a).

When the explicit interaction mode is used in *eSkel*, the user can control the timing of communications within the spatial constraints imposed by the skeleton. This means, a stage function does not need to receive all its input through its parameters and it does not necessarily pass output as a return value. Through direct calls to the *eSkel* functions `Give` and `Take`, new input can be retrieved or new output can be sent at any time within a stage function. For the skipping optimisation, we need to break the spatial constraints of the pipeline as well, i. e. any stage should be enabled to send an output not only to the next stage but also to the last.

5. The Gateway for Bridging between Web Services and MPI

In telecommunication terminology, a gateway is some hardware or software that addresses interconnection issues between systems using different protocols. For connecting our wavelet computation to a Web client, we developed a specially configured Web Service bridging between SOAP and an MPI-based pipeline. We call this service the gateway service and in the following, we explain its setup, which may be reused in other grid applications requiring an efficient pipeline implementation.

A particular feature of the gateway service is that it establishes a connection to an MPI-environment, which is not running inside the Web Service container and exhibits properties of its own. To maintain this connection the gateway service must store some *state data*, i. e. , data persisting the execution of single operations. The minimum state data required for the gateway service is already included in the configuration of the Pipeline HOC and consist in an array of output variables for holding results and the number of the TCP-port used for transferring application data between the MPI-environment and the Web Service.

While plain Web Services do not support state data at all, WSRF, as defined by OASIS [9], allows to bind a Web Service to state data. In order to use this feature, we deployed our HOC in the *globus-wsc-container* [13], which allows to run WSRF-compliant services written in C. The Globus middleware supports Web Services, whose interface descriptions include a *resource property document*. This document defines, in XML-Schema [15] format, the structure of the state data persisting during the execution of single service operations. For running the wavelet lifting algorithm, we specified the resource properties of the gateway service corresponding to the parameters of the application, i. e. , except the mandatory data described above, we declared one string property per stage function and one integer property giving the number of processes to be executed within the external MPI-environment.

The separation of the MPI-environment from the Web Service container and the use of an extra communication channel (TCP in our implementation) inside the gateway is a necessity. Web Service containers like Globus are parallel applications which can process multiple requests simultaneously via multithreading. Therefore, they must not be run within a multiprocess environment like MPI themselves, which would lead to running one extra container per MPI-process, making resource sharing unfeasible and furthermore resulting in a process management overhead.

Our gateway service assembles a `command`-string wherein a platform dependent prefix holds the path to the MPI-installation directory, followed by `mpirun -np # pipeline` with the #-parameter reflecting the `numProcessors` resource property. Upon request of the `init`-operation, the service launches the MPI-program by calling `system(command)`. The MPI-program `pipeline` starts by opening a TCP server socket which accepts input in the form of number series or images. This connection is established only by process 0, i. e. , processes with a higher rank wait until all input has arrived and is scattered amongst them.

6. Conclusion and Future work

This paper describes a high-level abstraction over native technologies on the grid using a Higher-Order Component (HOC). We implemented the lifting algorithm via sequential pipeline stages and applied it to multiple independent tasks in parallel. The choice of MPI was motivated by the fine grain of the computations in this application, which are not eligible to be dispersed across the grid. We implemented a Pipeline HOC allowing for local parallelism and for remote access. This HOC was customised for an application of the discrete wavelet transform. We also proposed a solution to avoid portability problems in the grid environments, where parameters must be exchangeable

between different software components. When a well-defined format is used for representing parameters, even the presence of multiple protocols and programming languages within a single system does not put insuperable barriers in the way of communication between services and clients. We identified *XPath* as a suitable format for customising HOCs rather than using executable code, which would restrict our HOC implementation to the use of a single programming language.

Customising components by transferring user-defined functions across the network is a new usage for the *XPath* language, applicable in multiple domains. In our future work, we plan to integrate *Apache Xalan* for this purpose with our HOC, which currently still requires to hard-wire the definitions of Section 4.1 to run a particular transform.

We also shown how the behavior of the pipeline skeleton can be optimised, when it is used in a lifting scheme application.

Finally, we plan to extend our HOC-based SOA approach to other skeleton implementations that use native machine code.

Acknowledgements: This work has been performed under the Project HPC-EUROPA (RII3-CT-2003-506079), with the support of the European Community - Research Infrastructure Action under the FP6 "Structuring the European Research Area" Programme and the EPSRC project Enhance (under grant number GR/S21717/01).

References

[1] Anne Benoit and Murray Cole. Two fundamental concepts in skeletal parallel programming. In P. Sloot V. Sunderam, D. van Albada and J. Dongarra, editors, *The International Conference on Computational Science (ICCS 2005) , Part II*, LNCS 3515, pages 764–771. Springer Verlag, 2005.

[2] Murray I. Cole. *Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computation*. Pitman, 1989.

[3] Murray I. Cole. Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. In *Parallel Computing 30*, pages 389–406, 2002.

[4] Thomas Erl. *Service-Oriented Architecture : A Field Guide to Integrating XML and Web Services*. Prentice Hall PTR, 2004.

[5] Ian Foster. *Designing and Building Parallel Programs. Concepts and Tools for Parallel Software Engineering.* Addison Wesley, 1995.

[6] Sergei Gorlatch and Jan Dünnweber. From grid middleware to grid applications: Bridging the gap with HOCs. In *Future Generation Grids*. Springer Verlag, 2005. to appear.

[7] Barbara Burke Hubbard. *The world according to wavelets.* A K Peters Ltd., Wellesley, MA, 1998. second ed.

[8] Arne Jensen and Anders la Cour-Harbo. *Ripples in mathematics: the discrete wavelet transform.* Springer Berlin, 2001.

[9] OASIS Technical Committee. WSRF: The Web Service Resource Framework, http://www.oasis-open.org/committees/wsrf.

[10] Apache Organization. Apache xalan. http://xml.apache.org/xalan-c.

[11] Heinz-Otto Peitgen and Peter H. Richter. *The Beauty of Fractals, Images of Complex Dynamical Systems*. Springer-Verlag New York Inc, June 1996.

[12] Wim Sweldens. The lifting scheme: A custom-design construction of biorthogonal wavelets. *Appl. Comput. Harmon. Anal.*, 3(2):186–200, 1996.

[13] The Globus Alliance. GT 4.0: C WS Core. http://www.globus.org/toolkit/docs/4.0/common/cwscore.

[14] Unicore Forum e.V. UNICORE-Grid. http://www.unicore.org.

[15] World Wide Web Consortium, W3C. The XML Schema definition language recommendation. http://www.w3.org/XML/Schema.

[16] World Wide Web Consortium, W3C. XML Path Language (XPath) 2.0. http://www.w3.org/TR/xpath20.

Scalable Farms

Michael Poldner [a], Herbert Kuchen [a]

[a]University of Münster, Department of Information Systems, Leonardo Campus 3,
D-48159 Münster, Germany

Algorithmic skeletons intend to simplify parallel programming by providing a higher level of abstraction compared to the usual message passing. Task and data parallel skeletons can be distinguished. In the present paper, we will consider several approaches to implement one of the most classical task parallel skeleton, namely the farm, and compare them w.r.t. scalability, overhead, potential bottlenecks, and load balancing. Based on experimental results, the advantages and disadvantages of the different approaches are shown.

1. Introduction

Today, parallel programming of MIMD machines with distributed memory is typically based on message passing. Owing to the availability of standard message passing libraries such as MPI [1] [11], the resulting software is platform independent and efficient. However, the programming level is still rather low and programmers have to fight against low-level communication problems such as deadlocks. Moreover, the program is split into a set of processes which are assigned to the different processors. Like an ant, each process only has a local view of the overall activity. A global view of the overall computation only exists in the programmer's mind, and there is no way to express it more directly on this level.

Many approaches try to increase the level of parallel programming and to overcome the mentioned disadvantages. Here, we will focus on *algorithmic skeletons*, i.e. typical parallel-programming patterns which are efficiently implemented on the available parallel machine and usually offered to the user as higher-order functions, which get the details of the specific application problem as argument functions (see e.g. [3,4,9]). [6] contains links to virtually all groups and projects working on skeletons.

In our framework, a parallel computation consists of a sequence of calls to skeletons. Several implementations of algorithmic skeletons are available. They differ in the kind of host language used and in the particular set of skeletons offered. Since higher-order functions are taken from functional languages, many approaches use such a language as host language [7,13,18]. In order to increase the efficiency, imperative languages such as C and C++ have been extended by skeletons, too [2,3,9,10].

Depending on the kind of parallelism used, skeletons can be classified into *task parallel* and *data parallel* ones. In the first case, a skeleton (dynamically) creates a system of communicating processes by nesting predefined process topologies such as `pipeline`, `farm`, `parallel composition`, `divide&conquer`, and `branch&bound` [1,4,5,7,9,12]. In the second case, a skeleton works on a distributed data structure, performing the same operations on some or all elements of this data structure. Data-parallel skeletons, such as `map`, `fold` or `rotate` are used in [2,3,7–9,13].

Moreover, there are implementations offering skeletons as a library rather than as part of a new programming language. The approach described in the sequel is based on the skeleton library intro-

[1]We assume some familiarity with MPI and C++.

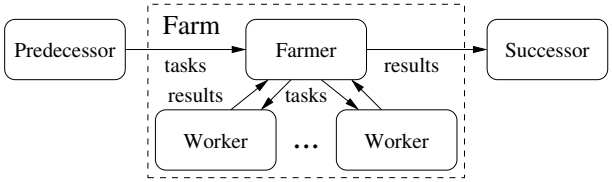


Figure 1. Farm.

duced in [12,14,15] and on the corresponding C++ language binding. Skeletons can be understood as domain-specific languages for parallel programming.

Our library provides task as well as data parallel skeletons, which can be combined based on the *two-tier model* taken from P^{3}L [9]. In general, a computation consists of nested task parallel constructs where an atomic task parallel computation can be sequential or data parallel. Purely data parallel and purely task parallel computations are special cases of this model. An advantage of the C++ binding is that the three important features needed for skeletons, namely higher-order functions (i.e. functions having functions as arguments), partial applications (i.e. the possibility to apply a function to less arguments than it needs and to supply the missing arguments later), and parametric polymorphism, can be implemented elegantly and efficiently in C++ using operator overloading and templates, respectively [14,16,19].

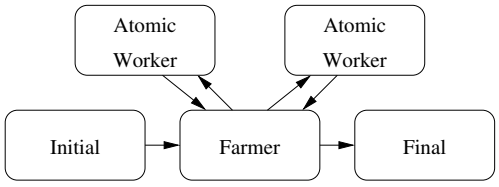
In the present paper, we will focus on task-parallel skeletons in general and on the well-known *farm* skeleton in particular. Conceptually, a farm consists of a *farmer* and several *workers*. The farmer accepts a sequence of tasks from some predecessor process and propagates each task to a worker. The worker executes the task and delivers the result back to the farmer who propagates it to some successor process (which may be the same as the predecessor). This specification suggests a straightforward implementation leading to the process topology depicted in Fig. 1. The problem with this simple approach is that the farmer may become a bottleneck, if the number of workers is large. Another disadvantage is the overhead caused by the propagation of messages. Consequently, it is worth considering different implementation schemes avoiding these disadvantages. In the present paper, we will consider a variant of the classical farm where the farmer is divided into a *dispatcher* and a *collector* as well as variants where these building blocks have (partly) been omitted.

The rest of the paper is organized as follows. In Section 2, we show how task-parallel applications can be implemented using the task-parallel skeletons provided by our skeleton library. The considered variants of the farm skeleton are presented in Section 3. Section 4 contains experimental results for the different approaches. Finally, in Section 5 we conclude and discuss related work. Moreover, we point out future extensions.

2. Task-parallel Skeletons

Our skeleton library offers data parallel and task parallel skeletons. Task parallelism is established by setting up a system of processes which communicate via streams of data. Such a system is not arbitrarily structured but constructed by nesting predefined process topologies such as farms and pipelines. Moreover, there are skeletons for parallel composition, branch & bound, and divide & conquer. Finally, it is possible to nest task and data parallelism according to the mentioned two-tier model of P^{3}L, which allows atomic task parallel processes to use data parallelism inside [9]. Here, we will focus on task-parallel skeletons in general and on the farm skeleton in particular.

In a farm, a farmer process accepts a sequence of inputs and assigns each of them to one of several




```
#include "Skeleton.h"

static int current = 0;
static const int numOfWorkers = 2;

int* init() { if (current++ < 100000) return &current;
            else return NULL;}

int add(int x, int y) { return x + y; }

void fin(int n) { cout << "result: " << n << endl; }

int main(int argc, char **argv){
    InitSkeletons(argc,argv);

    // step 1: create a process topology (using C++ constructors)
    Initial<int>     initial(init);
    Atomic<int,int>  atomicWorker((curry(add)(i),1));
    Farm<int,int>    farm(atomicWorker,numOfWorkers);
    Final<int>       final(fin);
    Pipe             pipeline(initial,farm,final);

    // step 2: start the system of processes
    pipeline.start();

    TerminateSkeletons();}
```

Figure 2. Task parallel example application.

workers. The parallel composition works similar to the farm. However, each input is forwarded to every worker. A pipeline allows tasks to be processed by a sequence of stages. Each stage handles one task at a time, and it does this in parallel to all the other stages.

Each task parallel skeleton has the same property as an atomic process, namely it accepts a sequence of inputs and produces a sequence of outputs. This allows the task parallel skeletons to be arbitrarily nested. Task parallel skeletons like pipeline and farm are provided by many skeleton systems, see e.g. [5,9].

In the example in Fig. 2, a pipeline of an initial atomic process, a farm of two atomic workers, and a final atomic process is constructed. In the C++ binding, there is a class for every task parallel skeleton. All these classes are subclasses of the abstract class `Process`. A task parallel application proceeds in two steps. First, a process topology is created by using the constructors of the mentioned class. This process topology reflects the actual nesting of skeletons. Then, this system of processes is started by applying method `start()` to the outermost skeleton. Internally, every atomic process will be assigned to a processor. For an implementation on top of SPMD, this means that every processor will dispatch depending on its rank to the code of its assigned process. When constructing an atomic process, the argument function of the constructor tells how each input is transformed into an output value. Again, such a function can be either a C++ function or a partial application. In Fig. 2, worker i adds i to all inputs. The initial and final atomic processes are special, since they do not consume inputs and produce outputs, respectively.

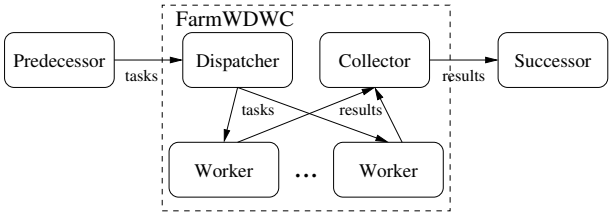


Figure 3. Farm with separate dispatcher and collector.

3. The Farm Skeleton

As pointed out in the introduction, a straightforward implementation of the farm skeleton could be based on the process topology depicted in Fig. 1. It has the advantage that the farmer knows which workers have returned the results of their tasks and are hence idle. Thus, the farmer can forward incoming tasks to the idle workers. However, this approach has the disadvantage that it causes substantial overhead due to the messages which have to be exchanged between farmer and workers. Moreover, the farmer might become a bottleneck, if the number of workers is large. In this case, the farmer will not be able to keep all workers busy, leading to wasted workers. The amount of workers which the farmer can keep busy depends on the sizes of the tasks and the sizes of the messages the farmer has to propagate. For small tasks and large messages, only very few workers can be kept busy.

If on the other hand, there are few workers (with large tasks), the farmer will partly be idle. This problem can be solved by mapping a worker to the same processor as the farmer. Thus, we will not consider this problem further. In general, the aim is to keep all processors as busy as possible and to avoid a waste of resources.

Let us point out that implementing this apparently simple farm skeleton is not as easy as it might seem. The interested reader may have a look at our implementation [17]. Firstly, the process topology is obviously cyclic (see Fig. 1). Thus, one has to be very careful to avoid deadlocks. On the other hand, one has to make sure that the farmer reacts as quickly as possible on newly arriving tasks and on workers delivering their results. For an implementation based on MPI, this means that the simpler synchronous communication has to be replaced by the significantly more complex non-blocking asynchronous communication using `MPI_Wait`. Moreover, special control messages are needed besides data messages in order to stop the computation cleanly at the end. This leads to a quite complicated protocol, which has to be supported by every skeleton. Thus, an obvious advantage of using skeletons is that the user does not have to invent the wheel again and again, but can readily use the typical patterns for parallel programming without worrying about deadlocks, stopping the computation cleanly, or overlapping computation and communication properly.

3.1. Farm with Dispatcher and Collector

A first approach to reduce the load of an overloaded farmer is to split it into a *dispatcher* of work and an independent *collector* of results as depicted in Fig. 3. This variant is implemented in P^3L [9].

In case that distributing tasks needs as much time as collecting results, the farmer can serve twice as many workers as with the previous approach. If, however, distributing tasks is much more work than collecting results or vice versa, little has been gained, since now the dispatcher or the collector will quickly become the bottleneck, while the other will partly be idle. The amount of required messages is unchanged and the corresponding overhead is hence preserved. A disadvantage of this farm variant is that the dispatcher now has no knowledge about the actual load of each worker. Thus,

he has to divide work based on some load independent scheme, e.g. cyclically or by random selection of a worker. Both may lead to an unbalanced distribution of work. However for a large number of tasks, one can expect that the load is roughly balanced. The blind distribution of work could be avoided by sending work requests from idle workers to the dispatcher. But then the dispatcher would have to process as many messages as the farmer in the original approach, and the introduction of a separate collector would be rather useless.

3.2. Farm with Dispatcher

The previous approach can be improved by omitting the collector and by sending results directly from the workers to the successor of the farm in the overall process topology (see Fig. 4). This eliminates the overhead of propagating results. Moreover, the omitted collector can no longer be a potential bottleneck.[2]

3.3. Farm without Dispatcher and Collector

After omitting the collector, one can consider omitting the dispatcher as well (see Fig. 5). In fact, this is possible, provided that the predecessor(s) of the farm in the overall process topology assume(s) the responsibility to distribute its/their tasks directly to the workers. As in the previous subsections, this distribution has to be performed based on a load independent scheme, e.g. cyclically or by random selection.

This approach reduces the overhead for the propagation of messages completely. Moreover, it omits another potential source of a bottleneck, namely the dispatcher. Of course, this new variant of the farm skeleton can be arbitrarily nested with other skeletons. For instance, it is possible to construct a pipeline of the such farms, as depicted in Fig. 5. In such a situation where n workers of the first farm communicate with m workers of the second, it is important to ensure that not all of them start with the same destination. If using a cyclic distribution scheme, worker i could e.g. assign its first task to worker $\lfloor i/m \rfloor$ of the second farm. If the destination is randomly chosen, it has to be ensured that all the random number generators start with a different initial value.

A farm without dispatcher but with collector does not seem to make sense, and it is not considered here.

4. Experimental Results

We have tested the different variants of the farm skeleton for small and big tasks as well as for tasks of variable sizes. A small task simply consists of computing the square of the input, a big task performs two million additions, and the tasks of variable sizes execute $n!$ iterations for some $1 \leq n \leq 10$.

[2] It may however happen that the successor now gets a bottleneck. In this case, the overall process topology needs to be adapted.

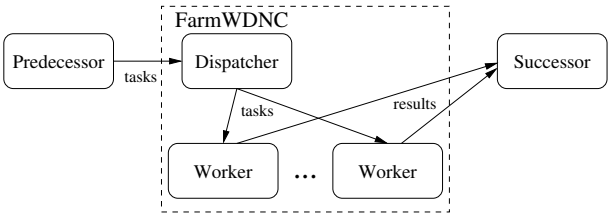


Figure 4. Farm with dispatcher; no collector is used.

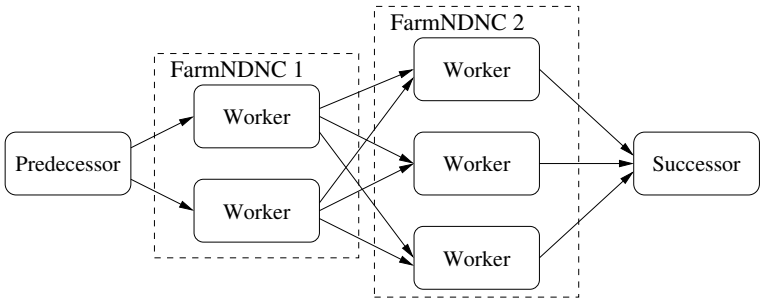


Figure 5. Pipeline of two farms without dispatcher and collector.

The experiments have been carried out on the IBM cluster at the University of Münster. This machine [20] has 94 compute nodes, each with a 2.4 GHz Intel Xeon processor, 512 KB L2 Cache, and 1 GB memory. The nodes are connected by a Myrinet and running RedHat Linux 7.3 and the MPICH-gm implementation of MPI.

For large tasks, all variants are able to keep the workers busy, as one would expect. Thus, all variants need roughly the same amount of time to execute the tasks, as can be seen in Fig. 6 a). However, just comparing the runtimes is not fair, since the different farms need different numbers of processors (unless farmer, dispatcher and collector share the same processor with one worker as mentioned above).Taking this into account, we see that the farmNDNC is the best, followed by the farmWDNC (see Fig. 6 b).

When considering tasks with (strongly) varying sizes (Fig. 7), we note that all approaches are able to keep a small number of workers (here up to 4) busy. If we add more workers, all approaches reach a point where they are no longer able to employ the additional workers. Beyond this point the runtime remains constant, independently of the number of additional workers. As expected, the original farm reaches this situation more quickly than the farmWDWC, the farmWDNC, and the farmNDNC, in this order. Moreover, farms which produce less overhead need less runtime when reaching the limit. When taking into account the number of processors used (rather than the number of workers), the advantages of the farmWDNC and, in particular, the farmNDNC become even more apparent (see Fig. 7 b). Interestingly, the behavior of the different farm variants does not depend significantly on the distribution scheme. Cyclic distribution and random distribution lead to almost identical runtimes. This is due to the fact that the number of tasks was large and the load of the workers has hence been balanced over time (Fig. 7 b). For small numbers of tasks the behavior may depend heavily on the actual mapping of tasks to workers.

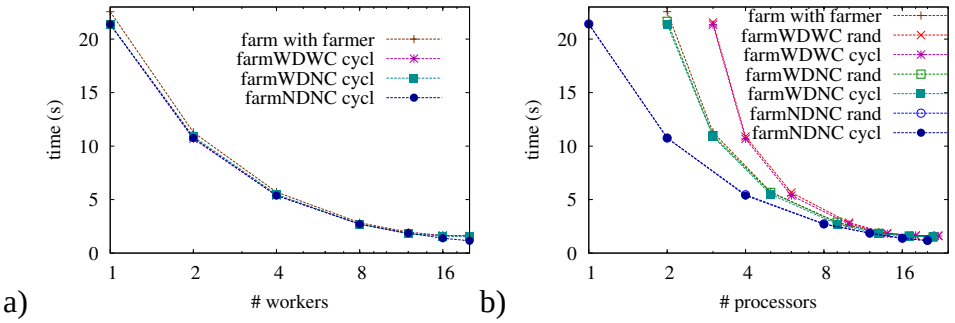


Figure 6. Farm variants with big tasks.

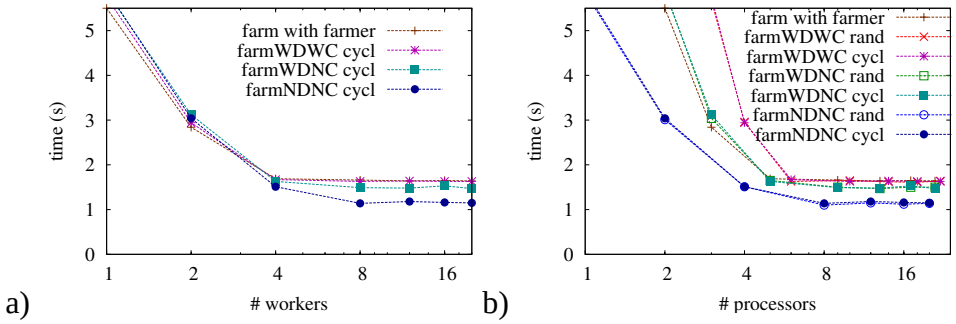


Figure 7. Farm variants with tasks of variable sizes.

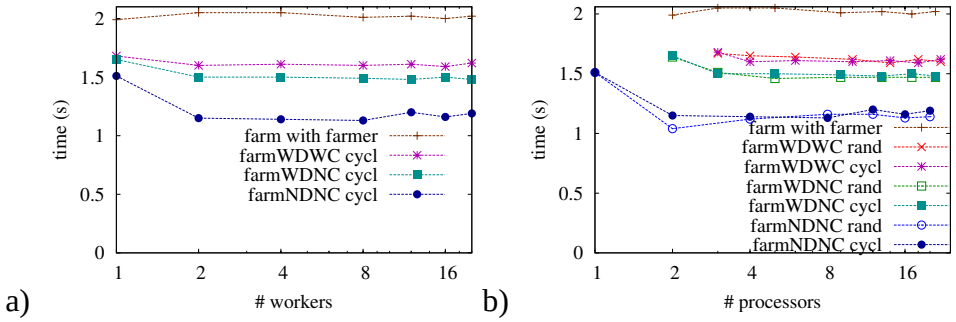


Figure 8. Farm variants with small tasks.

It may be surprising that even the farmNDNC is not able to keep an arbitrary amount of workers busy, although it has no farmer or dispatcher which might get a bottleneck. The reason is that the predecessor(s) of the farm are now responsible for the distribution of tasks to the workers. Unless the predecessor is itself a large farm, it will eventually become a bottleneck as observed in Fig. 7.

For small tasks, it is not worthwhile to employ any worker. The process delivering the small tasks should better execute them itself. If we nevertheless use a farm, it is clear that it is not possible to keep even a single worker busy. All farm variants require hence a more or less constant runtime independent of the number of workers (see Fig. 8). This situation corresponds to the one in the previous experiment when reaching the limit of useful workers. The roughly constant runtimes are not the same for all the skeletons but depend on the overhead caused by the considered variant.

5. Conclusions, Related Work, and Future Extensions

We have considered alternative implementation schemes for the well-known farm skeleton. Besides the classical approach where a farmer distributes work and collects results, we have considered variants where the farmer has been divided into a dispatcher and collector. Moreover, we have investigated variants where the collector and dispatcher have (partly) been omitted. In case of the variant without dispatcher, the predecessor of the farm in the overall process topology is responsible for the distribution of tasks to the workers. As our experimental results and our analysis show, the farm only consisting of workers is the best in terms of scalability and low overhead. It is clearly superior to the farms with dispatcher (and possibly collector). For a large number of tasks, it is also better than the classical farm, where a farmer distributes work. For a very small number of tasks, the classical farm might have advantages in some cases, since it is the only one which takes the actual load of workers

into account when distributing work.

We are not aware of any previous systematic analysis of different implementation schemes for farms in the literature. In the different skeleton projects, typically one technique has been chosen. In the first version of our skeleton library, there was just the classical farm. P^3L [9] uses the farmWDWC while eSkel [5] uses the classical farm with farmer.

As future work, we intend to investigate alternative implementation schemes for other skeletons, e.g. divide & conquer and branch & bound.

References

[1] Alba, E., Almeida, F., et al.: MALLBA: A Library of Skeletons for Combinatorial Search. Euro-Par'02. LNCS 2400, 927-932. Springer Verlag. 2002.

[2] Botorog, G.H., Kuchen, H.: Efficient Parallel Programming with Algorithmic Skeletons. In Bougé, L. et al., eds.: Euro-Par'96. LNCS 1123, 718-731. Springer-Verlag. 1996.

[3] Botorog, G.H., Kuchen, H.: Efficient High-Level Parallel Programming. Theoretical Computer Science 196, 71-107. 1998.

[4] Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press. 1989.

[5] Cole, M.: Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. Parallel Computing 30(3), 389-406. 2004.

[6] Cole, M.: The Skeletal Parallelism Web Page. http://homepages.inf.ed.ac.uk/mic/Skeletons/.

[7] Darlington, J., Field, A.J., Harrison T.G., et al: Parallel Programming Using Skeleton Functions. PARLE'93. LNCS 694, 146-160. Springer-Verlag. 1993.

[8] Darlington, J., Guo, Y., To, H.W., Yang, J.: Functional Skeletons for Parallel Coordination. In Hardidi, S., Magnusson, P.: Euro-Par'95. LNCS 966, 55-66. Springer-Verlag. 1995.

[9] Danelutto, M., Pasqualetti, F., Pelagatti S.: Skeletons for Data Parallelism in P^3L. In Lengauer, C., Griebl, M., Gorlatch, S.: Euro-Par'97. LNCS 1300, 619-628. Springer-Verlag. 1997.

[10] Foster, I., Olson, R., Tuecke, S.: Productive Parallel Programming: The PCN Approach. Scientific Programming 1(1) 51-66. 1992.

[11] Gropp, W., Lusk, E., Skjellum, A.: Using MPI. MIT Press. 1999.

[12] Kuchen, H., Cole, M.: The Integration of Task and Data Parallel Skeletons. Parallel Processing Letters 12(2), 141-155. 2002.

[13] Kuchen, H., Plasmeijer, R., Stoltze, H.: Efficient Distributed Memory Implementation of a Data Parallel Functional Language. PARLE'94. LNCS 817, 466-475. Springer-Verlag. 1994.

[14] Kuchen, H., Striegnitz, J.: Higher-Order Functions and Partial Applications for a C++ Skeleton Library. Joint ACM Java Grande & ISCOPE Conference, 122-130. ACM. 2002.

[15] Kuchen, H.: A Skeleton Library. In Monien, B., Feldmann, R.: Euro-Par'02. LNCS 2400, 620-629. Springer-Verlag. 2002.

[16] Kuchen, H.: Optimizing Sequences of Skeleton Calls. in D. Batory, C. Consel, C. Lengauer, M. Odersky (Eds.): Domain-Specific Program Generation. LNCS 3016, 254-273. Springer Verlag. 2004.

[17] Kuchen, H.: The Skeleton Library Web Pages. http://danae.uni-muenster.de/lehre/kuchen/Skeletons/.

[18] Skillicorn, D.: Foundations of Parallel Programming. Cambridge U. Press. 1994.

[19] Striegnitz, J.: Making C++ Ready for Algorithmic Skeletons. Tech. Report IB-2000-08. http://www.fz-juelich.de/zam/docs/autoren/striegnitz.html.

[20] ZIV-Cluster: http://zivcluster.uni-muenster.de/.

"Second-generation" skeleton systems

M. Danelutto[a]

[a]Dept. Computer Science – University of Pisa – Largo Pontecorvo 3 – 56127 Pisa – Italy

Algorithmical skeletons, as originally introduced in late '80s, evolved under the pressure of users, on the one side, and designers, on the other side. The formers asking for new features, the latter perceiving the limits of the skeleton approach to parallel programming and trying to overcome them. In this work, we discuss the features that have to be tackled in a "second-generation", mature skeleton system. Actually, we propose to extend the requirements stated in previous work by Cole. We outline how these features have been taken into account in two programming environments we are currently developing at the University of Pisa, and we make a synthetic comparison with other known skeleton environments.
Keywords: structured parallel programming, skeletons, macro data flow, coordination languages, adaptivity, heterogeneity.

1. Introduction

When algorithmical skeletons were first introduced in late '80 [13] the idea had an almost immediate success. Several research groups started research tracks on the subject and come up with different programming environments supporting algorithmical skeletons. Darlington's group first developed functional language embeddings of the skeletons [18] and then moved to FORTRAN [19]. Our group designed P3L, which is basically a sort of skeleton parallel C [16,7]. Kuchen started the work on Skil [10] and eventually produced a C++ Skeleton Library [22]. Serot designed Skipper [24,25], which exploits the macro data flow implementation model introduced in [14]. The original definition of skeleton programming environment given by Cole in his book [11] "*The new system presents the user with a selection of independent "algorithmic skeleton", each of which describes the structure of a particular style of algorithm, in the way in which "higher order functions" represent general computational frameworks in the context of functional programming languages. The user must describe a solution to a problem as an instance of the appropriate skeleton.*" was almost completely embraced by these groups. In particular, the scientific community accepted the idea of a fixed selection of independent skeletons. All the above-mentioned skeleton systems developed in the '90 only provide the programmer with a fixed set of skeletons.

The fixed, immutable skeleton set was in the meanwhile a source of power and of a source of weakness for the skeleton systems. It allowed efficient implementations to be developed but also did not allow programmers to express neither non standard parallelism exploitation patterns nor patterns even slightly different from the ones provided by the supplied skeletons. A partial solution to the unavailability of skeletons modeling specific parallel patterns came from the implementation of skeletons as libraries, whose mechanisms adopted to exploit parallelism was partially known, such as the ones discussed in [17] or [9]. In the former case, skeletons are provided as plain C function calls. The input data stream and the output data stream are implemented by plain Unix file descriptors that are accessible to the user. Therefore the programmer can program his own parallel patterns and make them interact with the predefined skeletons just writing/reading data to/from standard file (pipe,

[0]This work has been partially supported by Italian national FIRB project no. RBNE01KNFP *GRID.it* and by the Italian national strategic project *legge 449/97* No. 02.00640.ST97.

actually) descriptors. In the latter case, skeletons are provided as collective MPI operations. The programmer can access the MPI communicator executing the single parallel activity of the skeleton (e.g. one pipeline stage or a task farm worker) and can freely manage the processors allocated to the communicator. Explicit primitives allow the programmer to receive tasks from the skeleton input stream and to deliver results to the skeleton output stream. In both cases, a limited degree of freedom is left to the programmer to program his own parallelism exploitation patterns either outside or inside the ones modeled by skeletons. Despite being around since long time and despite the progress made in skeleton system design and implementation, the skeleton systems did not take off as expected. Nowadays, the skeleton system usage is actually restricted to small communities grown around the teams that actually develop the skeleton systems.

Cole focused very well the problem in his *manifesto* [12]. Here he states that four problems have to be taken into account and solved to allow skeletons to gain significant popularity: ❶ *'propagate the concept with minimal conceptual disruption"*, that is skeletons must be provided within existing programming environments without actually requiring the programmers to learn entirely new programming languages ❷ *"integrate ad-hoc parallelism"*, i.e. allow programmers to express parallel patterns not captured by the available skeleton set ❸ *"accommodate diversity"*, that is provide mechanisms to specialize skeletons, in all those cases where specialization does not radically change the nature of the skeleton, and consequently the nature of the implementation, and ❹ *"show the payback"*, i.e. demonstrate that the effort required to adopt a skeleton systems is immediately rewarded by some kind of concrete results: shorter design and implementation time of applications, increased efficiency, increased machine independence of the application code, etc. While the second and the third points are more specifically technical, the first and the last one are actually more "advertising oriented". All these points, however, have impacts on both the way the skeleton systems are designed and on the way they are implemented.

In addition, we also claim that another small set of problems have to be tackled: ❺ *support code reuse*, that is allow programmers to reuse with minimal effort existing sequential code ❻ *handle target architecture heterogeneity*, i.e. implement skeletons in such a way skeleton programs can be run on clusters/networks/grids hosting heterogeneous computing resources (different processors, different operating systems, different memory/disk configurations, etc.) ❼ *handle dynamicity*, i.e. implement in the skeleton support proper support to handle typical dynamic situations, such as those arising when non dedicated processing elements are used (e.g. peaks of load that impair load balancing strategies) or from sudden unavailability of processing elements (e.g. network faults, node reboot). The first point comes from our P3L experience. P3L [7], and its "industrial" successor SkIE [8], both allowed portions of sequential code written in C, C++ and FORTRAN 77 to be included in skeletons. SkIE also allowed High Performance Fortran to be used in the building blocks of skeletons (e.g. in pipeline stages or in task farm workers). Users greatly appreciated this feature that allows to wrap existing code with minor effort and to reuse all the huge, existing library of (optimized) sequential code. The second and third points actually come from our experience in grid programming systems. Within the GRID.it FIRB three year Italian national project [21], we developed a structured parallel programming environment targeting clusters, networks and grids and based on the skeleton programming methodology: ASSIST [29,1,28]. Grid systems are dynamic and heterogeneous by definition [20] and any programming environment targeting grid architectures must include proper techniques and algorithms to take care of these two important aspects, possibly in an automatic and transparent way [26]. Overall, the solutions to the set of problems stated above should be considered the basis of "second generation" skeleton systems. In other words, mature skeleton technology should efficiently address all these problems. In this perspective, here we want

```
import muskel.*;

public static void main(String [] args) {
   ...
  Compute stage1 = new Farm(new doSweep());             // first stage, process the input param set
  Compute stage2 = new PostProcess();                   // second stage: sequential postprocess
  Compute mainProgram = new Pipeline(stage1,stage2);    // main program is a two stage pipeline
  ParDegree parDegree = new ParDegree(5);               // required parallelism degree
  ApplicationManager manager =
                  new ApplicationManager(mainProgram); // instantiate the application manager
  manager.setContract(parDegree);                       // now set the performance contract
  manager.inputStream("input.dat");                     // tell were are input task data
  manager.outputStream("output.dat");                   // tell were results must be stored
  manager.eval();                                       // compute in parallel
  ...                                                   // any code processing the results here ...
}
```

Figure 1. Sample **muskel** code

to discuss two "second-generation" experiences of our group in Pisa, namely the **muskel** one [15] and the ASSIST one [29,1,28]. Both are programming environments based on the skeleton structured parallel programming concepts. The former being a plain Java library exploiting macro data flow implementation techniques [14] derived from these used in Lithium [5], the latter defining a new programming language, which is actually a coordination language that uses skeletons to model parallelism exploitation patterns. ASSIST exploits implementation techniques derived from the implementation template methodology developed in P3L and adopted by other skeleton frameworks [22].

2. muskel

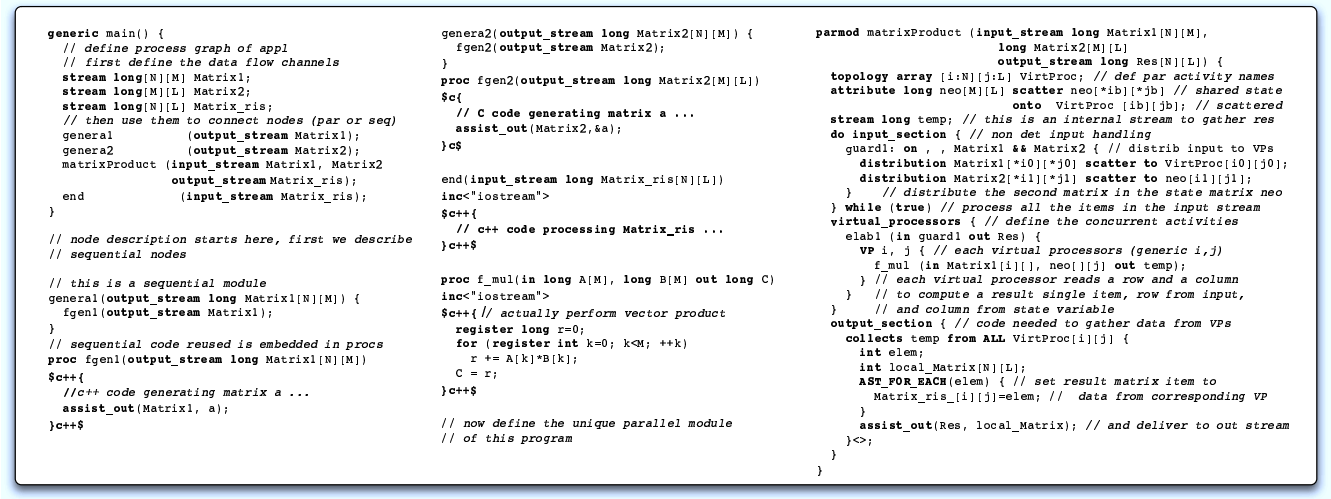
muskel (the name comes from the transliteration of μ-**skel**etons[1]) [15] is a full Java skeleton library providing user with usual stream parallel skeleton (pipelines, farms and arbitrary composition of farm and pipes). It is a compact, optimized subset of Lithium [5] and it has mainly being thought as a handy test bed to experiment new implementation strategies. Parallelism is exploited in **muskel** using plain java RMI. Remote interpreter objects are placed once and for all on all the processing elements possibly participating in the parallel computation. The skeleton library provides to automatically discover the processing elements where the remote interpreters have been placed and to recruit a suitable number of remote interpreters to schedule computations. Skeletons are implemented in **muskel** using macro data flow technology [14]: the skeleton program is translated into a data flow instruction graph. Instructions are fired when all the input tokens are available. A fireable instruction is simply scheduled for the execution on one of the available remote interpreters using RMI. Data flow instructions are actually "macro" data flow instructions. The user provides instruction functions as a parameter of the associate skeleton. In particular, sequential code to be used in these parameters is supplied as a `Compute` object, i.e. an object with an `Object compute(Object task)` method that returns a result after computing some sequential function on the input task object. In **muskel**, the skeleton program executed is not actually the one provided by the user: the skeleton program is first transformed to obtain its normal form as defined in [4] and then this normal form skeleton program is actually executed. A typical Java program using **muskel**

[1] we actually discovered recently that the Skeleton Library developed by Kuchen is called "muskel" on its web site. Here however, we use **muskel** to refer the Java library developed in Pisa and we refer to the other one as "Kuchen's Skeleton Library", to avoid name clashes

looks like the one in Figure 1. This program computes a two-stage pipeline, where the first stage is parallel (a task farm) and the second one is sequential. First the structure of the skeleton program is given. Then an `ApplicationManager` is instantiated and the performance contract (the parallelism degree, in this case) is passed to this manager along with the input and output files/streams. Eventually a single call is issued, the `manager.eval()` one, and this call takes care of all the steps needed to compute the skeleton program onto the stream of input tasks producing a stream of output results. In particular, the skeleton code is normalized and transformed into a macro data flow instruction graph, a discovery process is started and a number of remote interpreters congruent with the user supplied performance contract is contacted. The macro data flow instructions deriving from the skeleton code are staged to the remote interpreters. A thread is forked for each one of the remote interpreters recruited. The thread looks for fireable instructions in a *task pool* repository, delivers them to the associate remote interpreter and waits for the results of the computation. When results come back from the remote interpreter they are either delivered to the *result pool*, i.e. the place when they are taken to be delivered to the output stream, or reinserted in the proper target macro data flow instruction in the task pool. This macro data flow instruction can possibly become fireable. Immediately after the thread starts trying to fetch a new fireable instruction. In case a remote interpreter becomes unavailable (e.g. due to a network or to a node failure) the **`muskel`** application manager arranges to recruit a new one, if available. The task(s) left un-computed by the missing interpreter are put back to the task pool and they will be eventually re-scheduled to a different interpreter. Results achieved with **`muskel`** are very good both in terms of load balancing and in terms of fault tolerance and absolute performance/efficiency [15]. A minimal effort is required to experienced Java programmers to use **`muskel`**: basically, a small effort to implement the `Compute` interface in the existing application dependent code, plus the launch of the remote interpreter RMI objects on the available processing elements (both plain RMI and `rmid` versions of the remote interpreter are available) (thus addressing problem ❶). `FileInputStream` and `FileOutputStream` objects are passed to the manager to provide input task and retrieve output results. Therefore specialized parallel patterns can be programmed that interact with the existing skeleton (programs) via the streams, in the flavor of what happened in [17] with Unix file handles. Furthermore, recent improvements in the library [27] allow programmer to interact directly with the task/result pools. In particular, the user can fetch results from the result pool and can use them to build new (possibly fireable) macro data flow instructions to be inserted in the task pool. With such mechanisms, the programmer can either program his own macro data flow graphs or even implement completely new skeletons and add such skeletons to the library. Overall these two aspects allow both to integrate ad-hoc parallelism and to accommodate diversity (❷❸). The pay-back offered by **`muskel`** is shown by Figure 1, clearly evidencing the negligible amount of code needed to get a fully working, efficient parallel application (❹). Target architecture heterogeneity is handled naturally in **`muskel`** due to portability features of the JVM and RMI (❺). Dynamicity is handled in the `ApplicationManager`, where fault tolerance is also dealt with (❻). The only problem not actually solved is the support to code reuse (❼), as only Java code can be easily reused. The structure used to exploit parallelism heavily relies on serializability of code and it will be difficult to adapt to support C, FORTRAN or even C++ code reuse.

3. ASSIST

ASSIST (A Software development System based on Integrated Skeleton Technology) is a programming environment aimed at supporting parallel/distributed application development on clusters and networks of workstations as well as on grids. The environment implements the ASSIST co-



```
generic main() {
  // define process graph of appl
  // first define the data flow channels
  stream long[N][M] Matrix1;
  stream long[M][L] Matrix2;
  stream long[N][L] Matrix_ris;
  // then use them to connect nodes (par or seq)
  genera1          (output_stream Matrix1);
  genera2          (output_stream Matrix2);
  matrixProduct (input_stream Matrix1, Matrix2
                 output_stream Matrix_ris);
  end             (input_stream Matrix_ris);
}

// node description starts here, first we describe
// sequential nodes

// this is a sequential module
genera1(output_stream long Matrix1[N][M]) {
  fgen1(output_stream Matrix1);
}
// sequential code reused is embedded in procs
proc fgen1(output_stream long Matrix1[N][M])
$c++{
  //c++ code generating matrix a ...
  assist_out(Matrix1, a);
}c++$
```

```
genera2(output_stream long Matrix2[N][M]) {
  fgen2(output_stream Matrix2);
}
proc fgen2(output_stream long Matrix2[M][L])
$c{
  // C code generating matrix a ...
  assist_out(Matrix2,&a);
}c$

end(input_stream long Matrix_ris[N][L])
inc<"iostream">
$c++{
  // c++ code processing Matrix_ris ...
}c++$

proc f_mul(in long A[M], long B[M] out long C)
inc<"iostream">
$c++{// actually perform vector product
  register long r=0;
  for (register int k=0; k<M; ++k)
    r += A[k]*B[k];
  C = r;
}c++$

// now define the unique parallel module
// of this program
```

```
parmod matrixProduct (input_stream long Matrix1[N][M],
                      long Matrix2[M][L]
                      output_stream long Res[N][L]) {
  topology array [i:N][j:L] VirtProc; // def par activity names
  attribute long neo[M][L] scatter neo[*ib][*jb] // shared state
                           onto  VirtProc [ib][jb]; // scattered
  stream long temp; // this is an internal stream to gather res
  do input_section { // non det input handling
    guard1: on , , Matrix1 && Matrix2 { // distrib input to VPs
      distribution Matrix1[*i0][*j0] scatter to VirtProc[i0][j0];
      distribution Matrix2[*i1][*j1] scatter to neo[i1][j1];
    }    // distribute the second matrix in the state matrix neo
  } while (true) // process all the items in the input stream
  virtual_processors { // define the concurrent activities
    elab1 (in guard1 out Res) {
      VP i, j { // each virtual processors (generic i,j)
        f_mul (in Matrix1[i][], neo[][j] out temp);
      } // each virtual processor reads a row and a column
    }   // to compute a result single item, row from input,
  }     // and column from state variable
  output_section { // code needed to gather data from VPs
    collects temp from ALL VirtProc[i][j] {
      int elem;
      int local_Matrix[N][L];
      AST_FOR_EACH(elem) { // set result matrix item to
        Matrix_ris_[i][j]=elem; //  data from corresponding VP
      }
      assist_out(Res, local_Matrix); // and deliver to out stream
    }<>;
  }
}
```

Figure 2. Sample ASSIST code

ordination language, which is actually a language that allows to express arbitrary process graph applications, were each node in the graph can either be sequential or parallel, and nodes communicate via data flow streams [29]. Parallel nodes can be expressed using the `parmod` skeleton. A parmod (a generic **par**allel **mod**ule) can have multiple input stream and output streams. Programmer can implement arbitrary non-deterministic control on the input streams as well as to generate an arbitrary number of output items on the output streams. A parmod defines a set of logically concurrent parallel activities. The keyword *logically* refers to the fact that the ASSIST compiler and run time completely take care of executing them on the set of available/required processing elements in a transparent and optimized way. Such logically parallel activities, referred to as *virtual processors* in the ASSIST jargon, can be named as multidimensional arrays or as 'anonymous". In the former cases, virtual processors are distinguished in the program by indexes: input data can be delivered to specific virtual processors, or they can be broadcasted/multicasted to the virtual processors (this is used to implement data parallel computations as well as very specific parallel skeletons/patterns). In the latter case, the parmod only specifies the number of parallel activities: all the parallel activities are equivalent and input data can only be delivered to a generic virtual processor for processing (this is exploited to implement task farms). State variables can be shared among the virtual processors. The owner computes rule holds in case the shared variables are scattered across the virtual processors: a vector state variable x scattered across a vector of virtual processors allows virtual processor i to read any value $x[j]$ but to write only the value $x[j]$. The code executed by virtual processors, as well as the code executed by the sequential nodes, can be specified using C, C++ and FORTRAN77, at the moment, and we have already experimented the possibility of using Java code too. Virtual processor computation is triggered by the availability of all the input data specified in the virtual processors code. Therefore data flow execution mode is assumed. Virtual processors activities can be iterated and the compiler and run time support provide to insert proper synchronization to avoid processing data relative to different iterations. Figure 2 shows an ASSIST program with two sequential processes generating each a stream of matrixes and a parmod node multiplying such matrixes exploiting data parallelism. The ASSIST parmod represents the major innovation of ASSIST with respect to previous skeleton systems developed at our group. The parmod represents a *generic* parallel module. By specializing this generic construct many par-

PROBLEMS	NOTABLE SOLUTIONS IN: *muskel*	*ASSIST*	*eSkel*	*Kuchen's Skelton Library*
minimal conceptual disruption ❶	plain Java lib		plain MPI collectives	plain C++ lib
ad-hoc parallelism ❷	macro data flow level accessible to user + access to streams	parametric parmod	protected MPI communicators for parallel skeleton building blocks	variety of combination of (data parallel) skeletons
accommodating diversity ❸	same as above	same as above	parametric skeleton calls	
show pay-back ❹	OO lib expressive power, fast application development	fairly fast application development, highly efficient object code	fast application development	OO lib expressive power, fast application development
code reuse ❺	Java	C C++ FORTRAN	C C++	C C++
heterogenity ❻	guaranteed by Java	compiler + run time	guaranteed by MPI †	guaranteed by MPI †
dynamicity ❼	application manager	module + application manager		

† data type handling is in charge to the programmer, i.e. he should not use plain `MPI_Byte` data messages.

Figure 3. Summary of solutions to problems ❶–❼ in several advanced skeleton environments

allel skeletons can be derived: pipelines and farms, map/forall and reduce with or without shared state. The implementation of parmod is highly optimized and both relies on a huge compilation process and on an optimized run time system the ASSISTlib. The price to pay is a somehow heavy language. Also, the possibility to express arbitrary graphs of parallel/sequential nodes is a notable step away from previous experiences. As **`muskel`**, ASSIST supports autonomic control of parallel modules and of the overall application [6]. A parmod can be executed in such a way that the user asks a given performance contract to be satisfied. In this case, the parmod automatically provides to keep the contract satisfied, if possible: exploiting the knowledge coming from the parmod analytic performance models the parmod control dynamically adapts the number of resource used to execute the parmod, in such a way the user supplied performance model is satisfied. Some partners of the national project GRID.it including CNR, the Italian National Research Council, and ASI, the Italian Space Agency, currently use ASSIST to develop different kinds of applications: graphics (isosurface detection), bioinformatics (protein folding), "social" applications (sea oil spill detection, landslip detection) and chemistry (ab-initio molecule simulation). The major ASSIST pay-back is given by the results achieved when implementing these complex, possibly multidisciplinary applications: the development time was drastically reduced with respect to the time needed to develop equivalent, traditional parallel applications (e.g. applications programmed using MPI) and the efficiency achieved is almost the ideal one, in most cases (❹). However, the need to learn a completely new and fairly untraditional parallel language does not help to propagate the skeleton concept with minimal conceptual disruption (❶). Ad hoc parallelism is de facto integrated, through the noticeable reconfigurability of parmod (❷). The same feature allows accommodating diversity upon specific programmer/applications needs (❸). Code reuse is supported (C, C++, FORTRAN code reuse is already supported and Java support is forthcoming, ❺), heterogeneity is handled (current ASSIST version produces code running on networks of mixed Linux/Intel and MacOSX/PowerPC machines, ❻) and dynamicity is handled via the module and application managers implementing autonomic QoS control (❼). Actually, there are much more interesting properties and features in ASSIST, that are not being considered here because not relevant to the second-generation skeleton discussion. ASSIST supports its own component model, as an example [2], and it also supports seamlessly interaction with both CORBA/CCM and Web Services world. The interested reader may refer to [3].

4. Conclusions

We discussed the solutions given to problems ❶–❼ stated in Sec. 1 by two skeleton programming environments currently being developed in Pisa. Figure 3 shows a comparison of notable solutions given by either these two environments or by eSkel and Kuchen's C++ skeleton library. This summary table points out two aspects. Some environments react to problem ❷ and ❸ by providing limited/controlled access to the implementation level, in such a way users can program their own parallelism exploitation patterns. eSkel does it by allowing users to program parallel activities within the single component of a skeleton (e.g. a pipe stage), while `muskel` allows to program parallel activities outside the skeletons but interacting with the skeletons (we are currently working to extend these `muskel` features, indeed). A solution to problems ❷ and ❸ should probably provide both these possibilities. As skeleton systems are more and more oriented to give the user the possibility of programming his own skeletons, solutions such as the one adopted in [23] to guarantee controlled accesses to the implementation framework are needed. The other aspect to consider is that a tradeoff has to be found between the number of parameters needed to specify a skeleton and the skeleton system expressive power. ASSIST provides a highly customizable parmod skeleton, but the learning curve needed to make an efficient use of it is not negligible. Other systems provide much more strict skeletons, but they also must release the constrain to leave the implementation layer invisible to users, in order to guarantee solutions to ❷ and ❸. Library implementation of skeleton systems seems to guarantee a better framework to support this idea than implementations providing a full programming language. However, some compile time solutions that demonstrated very efficient in the implementation of ASSIST, as an example, look like very hard to implement in a library. Therefore techniques combining some kind of just-in-time compiling with library skeleton implementation should probably be exploited. Last but not least, solutions to problem ❺ to ❼ are fundamental to the success of skeletons systems as they guarantee to preserve the investements made in sequential software development and to target a larger and more significant class of architectures. Both `muskel` and ASSIST experience demonstrated that the ability of adapting application execution to varying features of an heterogeneous target architecture is a key point in convincing a user to migrate to a skeleton programming environment.

References

[1] M. Aldinucci, S. Campa, P. Ciullo, M. Coppola, S. Magini, P. Pesciullesi, L. Potiti, R. Ravazzoloand M. Torquati, M. Vanneschi, and C. Zoccolo. The Implementation of ASSIST, an Environment for Parallel and Distributed Programming. In *Proc. of Intl. Conference EuroPar2003: Parallel and Distributed Computing*, number 2790 in LNCS. Springer, 2003.

[2] M. Aldinucci, S. Campa, M. Coppola, M. Danelutto, D. Laforenza, D. Puppin, L. Scarponi, M. Vanneschi, and C. Zoccolo. Components for High-Performance Grid Programming in GRID.it. In *Component modes and systems for Grid applications*, CoreGRID. Springer, 2005.

[3] M. Aldinucci, M. Coppola, S. Campa, M. Danelutto, M. Vanneschi, and C. Zoccolo. Structured Implementation of Component based GRID programming environments. In *FGG: Future Generation Grids*, CoreGRID. Springer Verlag, 2005.

[4] M. Aldinucci and M. Danelutto. Stream parallel skeleton optimisations. In *Proc. of the IASTED International Conference Parallel and Distributed Computing and Systems*, pages 955–962. IASTED/ACTA Press, November 1999. Boston, USA.

[5] M. Aldinucci, M. Danelutto, and P. Teti. An advanced environment supporting structured parallel programming in Java. *Future Generation Computer Systems*, 19(5):611–626, 2003. Elsevier Science.

[6] M. Aldinucci, A. Petrocelli, E. Pistoletti, M. Torquati, M. Vanneschi, L. Veraldi, and C. Zoccolo. Dy-

namic reconfiguration of grid-aware applications in ASSIST. In *11th Intl Euro-Par 2005: Parallel and Distributed Computing*, LNCS. Springer Verlag, 2005. to appear.

[7] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^3L: A Structured High level programming language and its structured support. *Conc. Practice and Experience*, 7(3):225–255, 1995.

[8] B. Bacci, M. Danelutto, S. Pelagatti, and M. Vanneschi. SkIE: a heterogeneous environment for HPC applications. *Parallel Computing*, 25:1827–1852, December 1999.

[9] A. Benoit, M. Cole, S. Gilmore, and J. Hillston. Flexible Skeletal Programming with eSkel. In *Proceedings of Euro-Par 2005: Parallel Processing*, LNCS. Springer Verlag, 2005. to appear.

[10] G. H. Botorog and H. Kuchen. Efficient high-level parallel programming. *Theoretical Computer Science*, 196(1–2):71–107, April 1998.

[11] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computations*. Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

[12] M. Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30(3):389–406, 2004.

[13] M. I. Cole. A "Skeletal" Approach to Exploitation of Parallelism. In C. R. Jesshope and K. D. Reinartz, editors, *CONPAR 88*, British Computer Society Workshop Series. Cambridge University Press, 1989.

[14] M. Danelutto. Dynamic Run Time Support for Skeletons. In E. H. D'Hollander, G. R. Joubert, F. J. Peters, and H. J. Sips, editors, *Proceedings of the International Conference ParCo99*, volume Parallel Computing Fundamentals & Applications, pages 460–467. Imperial College Press, 1999.

[15] M. Danelutto. QoS in parallel programming through application managers. In *Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-based processing*. IEEE, 2005. Lugano.

[16] M. Danelutto, R. Di Meglio, S. Orlando, S. Pelagatti, and M. Vanneschi. A methodology for the development and support of massively parallel programs. *FGCS*, 8(1–3):205–220, July 1992.

[17] M. Danelutto and M. Stigliani. SKElib: parallel programming with skeletons in C. In A. Bode, T. Ludwing, W. Karl, and R. Wismüller, editors, *Euro-Par 2000 Parallel Processing*, LNCS, No. 1900, pages 1175–1184. Springer Verlag, August/September 2000.

[18] J. Darlington, A. J. Field, P.G. Harrison, P. H. J. Kelly, D. W. N. Sharp, Q. Wu, and R. L. While. Parallel Programming Using Skeleton Functions. In M. Reeve A. Bode and G. Wolf, editors, *PARLE'93 Parallel Architectures and Langauges Europe*. Springer Verlag, June 1993. LNCS No. 694.

[19] J. Darlington, Y. Guo, H. W. To, Q. Wu, J. Yang, and M. Kohler. Fortran-S: A Uniform Functional Interface to Parallel Imperative Languages. In *Third Parallel Computing Workshop (PCW'94)*. Fujitsu Laboratories Ltd., November 1994.

[20] I. Foster and C. Kesselman (Editors). *The Grid 2 Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, December 2003.

[21] Grid.it community . The GRID.it home page, 2005. http://www.grid.it.

[22] H. Kuchen. A Skeleton Library. In *Euro-Par 2002, Parallel Processing*, number 2400 in LNCS, pages 620–629. "Springer" Verlag, August 2002.

[23] S. MacDonald, J. Anvik, S. Bromling, J. Schaeffer, D. Szafron, and K. Tan. From Patterns to Frameworks to Parallel Programs. *Parallel Computing*, 28(12):1663–1683, 2002.

[24] J. Sérot. Tagged-token data-flow for skeletons. *Parallel Processing Letters*, 11(4):377–392, Dec 2001.

[25] J. Sérot and D. Ginhac. Skeletons for parallel image processing : an overview of the SKiPPER project. *Parallel Computing*, 28(12):1785–1808, Dec 2002.

[26] D. Snelling and K-Jeffrey et al. Next Generation Grids 2 – Requirements and Options for European Grids Research 2005-2010 and Beyond, 2004. ftp://ftp.cordis.lu/pub/ist/docs/ngg2 eg final.pdf.

[27] S. Susini. Introduction of new features in a Java parallel programming environment *in italian*, June 2005. Final stage report of the "Diploma in Informatica", Dept. Computer Science, Univ. of Pisa.

[28] The ASSIST team. ASSIST home page, 2005. http://www.di.unipi.it/groups/architetture/Assist.html.

[29] M. Vanneschi. The Programming Model of ASSIST, an Environment for Parallel and Distributed Portable Applications. *Parallel Computing*, 12, December 2002.

Domain Decomposition and Skeleton Programming with OCamlP3l

F. Clément[a], V. Martin[a], A. Vodicka[a], R. Di Cosmo[b] P. Weis[c]

[a]Projet Estime, INRIA-Rocquencourt, B.P. 105, F-78153 Le Chesnay Cedex, France.

[b]Case 7014, Université de Paris 7, 2 place Jussieu, F-75251 Paris Cedex 05, France.

[c]Projet Cristal, INRIA-Rocquencourt, B.P. 105, F-78153 Le Chesnay Cedex, France.

When simulating large scale phenomena, it is natural to divide the domain of computation into subdomains. Then, besides the issue of the existence of a global solution to the coupled problem, arises the difficulty of its implementation. The OCamlP3l system provides a structured approach to parallel programming using skeletons and template based compilation techniques: designing and debugging is performed on a sequential version, and the parallel version is automatically deduced by a recompilation of the source program. We present the application of a domain decomposition method to a 3D flow problem around a deep underground nuclear waste disposal.

1. Introduction

This work deals with the simulation of flow and transport in porous media to study the feasibility of an underground nuclear waste disposal. Reliable simulations are crucial to forecast the behavior of contaminants in the geological layers, and thus should be computed in 3 dimensions on very long time scales (transport simulations over 10 million years), with very different length scales (going from the meter to 20 or 30 kilometers), involving possibly different physics. A natural way of treating such a large scale problem is to divide it into smaller subproblems, and then to couple them.

We restrict ourselves in this paper to the efficient simulation of stationary flow. The Darcy equations are solved by using a non-overlapping domain decomposition method that allows the treatment of non-matching grids. This coupling method is based on Robin interface conditions and was studied in [1]. This nonconforming domain decomposition method is very practical as it allows the separate meshing of the subdomains: for example, a local refinement around the underground storage, and the rest of the domain can be meshed with a coarser mesh, following the geological layers.

Code coupling and parallelism are a very demanding implementation task, especially when the codes to couple have been developed separately. The main difficulty is the fine tuning of the communications between the codes. Most of these aspects actually correspond to a sequence of generic basic tasks that should be automatically set up by a program, or better via compilation. This is where OCamlP3l enters the picture. Objective Caml (or OCaml) is a fast modern high-level functional programming language based on formal semantics. It is particularly well suited for the implementation of complex algorithms. As an example, the OCamlP3l system, see [3], provides a structured approach to parallel programming using skeletons and template based compilation techniques, see [6]. It brings all the piping capabilities we need to implement the communications, and moreover, it offers the parallelization for free: designing and debugging is driven on a sequential version, and the parallel version is automatically deduced by a simple recompilation of the same source program.

We present first the domain decomposition method in Section 2, then Section 3 is devoted to the OCamlP3l system and Section 4 to the implementation of the coupling algorithm, and finally in Section 5, we show numerical results for both extensibility tests and a realistic 3D flow computation.

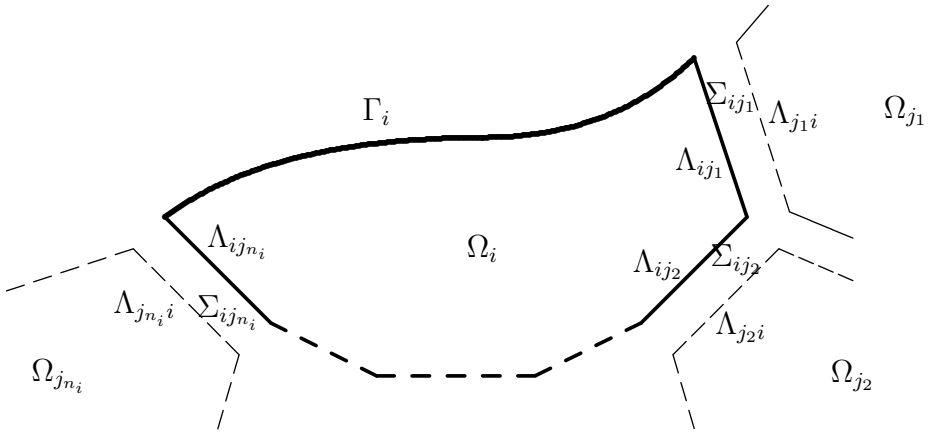


Figure 1. The subdomain Ω_i and its n_i neighbors Ω_{j_k}, $j_k \in \mathcal{N}_i$. The approximation spaces Λ_{ij_k} and $\Lambda_{j_k i}$ on each side of the interface Σ_{ij_k} are different.

2. Domain Decomposition

2.1. A model problem

Let Ω be a convex domain in $\mathbb{R}^d$, $d = 2$ or 3, and let $\Gamma = \partial\Omega$ be its boundary. We suppose that the flow in Ω is governed by a conservation equation together with Darcy's law relating the gradient of the pressure p to the Darcy velocity $\mathbf{u}$ via

$$\left\{ \begin{array}{rcll} \operatorname{div} \mathbf{u} & = & q & \text{in } \Omega \\ \mathbf{u} & = & -\mathbf{K}\boldsymbol{\nabla} p & \text{in } \Omega \\ p & = & \overline{p} & \text{on } \Gamma, \end{array} \right. \tag{1}$$

where $\mathbf{K}$ is the permeability tensor, q a source term and $\overline{p}$ the given pressure on the boundary Γ.

The domain Ω is decomposed into n non-overlapping subdomains Ω_i with $i \in I = \{1, 2, \ldots, n\}$, and we denote by $\Gamma_i = \partial\Omega_i \cap \Gamma$ the *external boundaries*. Let $\Sigma_i = \partial\Omega_i \backslash \Gamma$ be the *internal boundary* of Ω_i, and let $\Sigma = \bigcup_{i \in I} \Sigma_i$ be the geometric *structure* of the decomposition. Then, we can define the geometric *interface* between neighboring subdomains Ω_i and Ω_j as $\Sigma_{ij} = \Sigma_{ji} = \Sigma_i \cap \Sigma_j$. We denote the number of neighbors of subdomain Ω_i by n_i and the set of their indices by $\mathcal{N}_i = \{j_1, j_2, \ldots, j_{n_i}\}$. See Figure 1 for an illustration in the 2D case. The set of all couples of neighbors, called *connectivity table*, is denoted by $\mathcal{N} = \{(i, j)/i \in I, j \in \mathcal{N}_i\}$, it is naturally ordered by the lexicographic order on $\mathbb{N}^2$. Let s be the permutation involution on $\mathcal{N}$ defined by $s(i, j) = (j, i)$.

2.2. A fixed point formulation

Let $\mathcal{T}_i^h$ be a conforming finite element partition of the subdomain Ω_i. We denote by $\mathcal{S}_{ij}^h$ the trace of $\mathcal{T}_i^h$ on Σ_{ij}. In general, $\mathcal{S}_{ij}^h \neq \mathcal{S}_{ji}^h$ when the meshes $\mathcal{T}_i^h$ and $\mathcal{T}_j^h$ do not match at the interface Σ_{ij}.

Let $\Lambda_{ij} = P_0(\mathcal{S}_{ij}^h)$ the space of facewise, or edgewise in 2D, constant functions on $\mathcal{S}_{ij}^h$ and set

$$\Lambda_i = \bigoplus_{j\in\mathcal{N}_i} \Lambda_{ij}, \qquad \Lambda = \bigoplus_{i\in I} \Lambda_i = \bigoplus_{(i,j)\in\mathcal{N}} \Lambda_{ij},$$
$$\tilde{\Lambda}_i = \bigoplus_{j\in\mathcal{N}_i} \tilde{\Lambda}_{ij} = \bigoplus_{j\in\mathcal{N}_i} \Lambda_{ji}, \quad \tilde{\Lambda} = \bigoplus_{i\in I} \tilde{\Lambda}_i = \bigoplus_{(i,j)\in\mathcal{N}} \Lambda_{ji} = \bigoplus_{(i,j)\in s(\mathcal{N})} \Lambda_{ij}.$$

The global problem (1) can be rewritten as a transmission problem which states the same equations on each subdomain together with transmission conditions expressing continuity of the pressure and of the normal velocity across each interface Σ_{ij}. These transmission conditions can be rewritten as mixed Robin conditions with Robin coefficients $\alpha_{ij} > 0$ and $\alpha_{ji} > 0$ on each interface Σ_{ij}. Using a mixed finite element method with hybrid Lagrange multipliers, see [7], the discrete approximation of the local elliptic subproblems with Robin conditions take of the form

$$L_i v_i = (q_i, g(v_i)) \tag{2}$$

where v_i summarizes the velocity, pressure and Lagrange multiplier unknowns, and the function g expresses the Robin conditions on all the interfaces of the subdomain Ω_i.

Then, a simple method to solve all subproblems (2) for $i \in I$ is to solve iteratively the following fixed point problem: given $\lambda^0 \in \Lambda$, for $k \geq 0$, compute until convergence $\lambda_i^{k+1} = g(v_i^{k+1})$ where v_i^{k+1} is the solution of the local linear system

$$L_i v_i^{k+1} = (q_i, \lambda_i^k). \tag{3}$$

One can notice that matrix L_i is non-symmetric, hence when using an iterative method to solve (3), it is advisable to accelerate the convergence with a non-symmetric Krylov method such as BiCGStab or GMRes.

2.3. Interface operators

We introduce discrete Robin-to-Robin operators S_{iq_i} in each subdomain Ω_i defined by:

$$S_{iq_i} : \lambda_i \in \Lambda_i \longmapsto \mu_i = g(v_i) \in \Lambda_i \tag{4}$$

where v_i of the solution of the *inner* linear system

$$L_i v_i = (q_i, \lambda_i). \tag{5}$$

For all neighbors Ω_i and Ω_j, $(i,j) \in \mathcal{N}$, let $P_{i\to j}$ be the L^2-projection operator from $L^2(\Sigma_{ij})$ onto $\tilde{\Lambda}_{ij} = \Lambda_{ji}$. For all subdomain Ω_i, $i \in I$, let P_i be the tensor product $\bigotimes_{j\in\mathcal{N}_i} P_{i\to j}$. And let R_i (resp. $\tilde{R}_i$) be the restriction operator from Λ onto Λ_i (resp. from $\tilde{\Lambda}$ onto $\tilde{\Lambda}_i$).

Finally, we define the global operators

$$S_q = \sum_{i\in I} \tilde{R}_i^\top S_{iq_i} R_i : \Lambda \longrightarrow \Lambda \quad \text{and} \quad P = \sum_{i\in I} \tilde{R}_i^\top P_i R_i : \Lambda \longrightarrow \tilde{\Lambda} \tag{6}$$

and we set

$$A = \mathrm{Id}_\Lambda - sPS_0 \quad \text{and} \quad b = sPS_q(0). \tag{7}$$

With all these notations, solving the initial problem (1) is equivalent to find $\lambda \in \Lambda$ solution to the *outer* linear system

$$A\lambda = b. \tag{8}$$

Once again, since the matrix A is non-symmetric, it is advisable to accelerate the convergence of the resolution of (8) with a non-symmetric Krylov method such as BiCGStab or GMRes.

3. Skeleton Programming with OCamlP3l

Caml is a strongly-typed functional programming language from the ML (Meta Language) family. OCaml (Objective Caml) is an open source implementation of Caml developed at INRIA[1]. P3L (Pisa Parallel Programming Language) is a structured parallel programming language developed at the Department of Computer Science of the University of Pisa. It is based upon skeleton/templates and allows parallel programs to be developed composing a small set of primitive parallel forms[2]. OCamlP3l is a parallel programming system based on OCaml and P3l languages, providing seamless integration of parallel programming and functional programming and advanced features like sequential logical debugging of parallel programs and strong typing, useful both in teaching parallel programming and in the building of full-scale applications[3].

We briefly present now the OCamlP3l system, and the reader can refer to [2] for more details.

3.1. Three semantics

A distinctive feature of the OCamlP3l system, among all skeleton-based systems, is that the semantics of the skeletons is not hard-wired: the system allows the user to compile his code without any source modification using three possible semantics, i.e. three implementations of the skeletons. The *sequential semantics* produces an executable that can run on a single machine, as a single process, and easily debugged using standard techniques and tools for sequential programs. The *parallel semantics* produces a generic SPMD program that can be deployed on a parallel machine, a cluster, or a network of workstations. The *graphical semantics* produces a program that displays a picture of the parallel computational network that is deployed when running the parallel version.

A key issue in the further development of OCamlP3l will be the proof of the adequation theorem stating the agreement between sequential and parallel executions: under reasonable assumptions, for any user program the two semantics should produce exactly the same result. Hence, the user has only to debug the sequential version.

3.2. Skeletons combinators

The OCamlP3l skeletons are compositional: the skeletons are combinators that form an algebra of functions and functionals called the *skeleton language* that define the parallel behavior of programs. More precisely, a skeleton is a *stream processor*, i.e. a function that transforms an input stream of incoming data into an output stream of outgoing data.

In version 2.0 of OCamlP3l, the eight combinators pertain to five kinds:

- the *task parallel* skeletons `farm` and *pipeline* model the parallelism of independent processing activities relative to different input data.
- the *data parallel* skeletons `mapvector` and `reducevector` model the parallel computation of different parts of the same input data.
- the *data interface* skeletons `seq` and `parfun` provide (dual) injection and projection between sequential and parallel worlds.
- the *parallel execution scope delimiter* skeleton `pardo` must encapsulate all the code that invokes `parfun`'s.
- the *control* skeleton `loop` provides the repetitive execution of a given skeleton expression.

[1] see `http://caml.inria.fr/`.
[2] see `http://www.di.unipi.it/~susanna/p3l.html`.
[3] see `http://www.ocamlp3l.org/`.

The `mapvector` combinator computes in parallel a function over all the components of *vector* data items of the input stream. The expression `mapvector(`f`,`k`)` computes $(f(x_i^1), \ldots, f(x_i^n))$ over all (vector) data items $x_i = (x_i^1, \ldots, x_i^n)$ by having k independent processes that compute f over different components of the vector.

The `seq` combinator converts a sequential function into a node of the parallel computational network.

The `parfun` combinator converts a parallel computational network into a sequential stream processing function.

See [2] for the description of the other skeletons not used in the present application. All skeleton combinators allow for global or local initialization, meaning that independent processes will share, or not, their initialization data.

4. Implementation of the code coupling

4.1. Code to couple

The code to be (self-)coupled is the C++ solve_on_a_subdomain code that inputs the name of the file describing the 3D mesh associated with a subdomain, e.g. Ω_i, for $i \in I$. It reads this file from the disk and enters an infinite loop waiting for a keyword:

When given the keyword `"init"`, it computes a sparse LU factorization of the matrix L_i used to solve the inner linear systems (5), then computes and outputs $S_{iq_i}0_i$ needed for the computation of the right-hand side b given by (7).

When given the keyword `"loop"`, it inputs λ_i, then computes and outputs $S_{i0_i}\lambda_i$ needed for the computation of the matrix-vector product.

When given the keyword `"final"`, it inputs $\lambda_i^\star$, then computes and writes on the disk the solution $v_i^\star$ to the problem (5) associated with $S_i q_i \lambda_i^\star$.

It needs redirection of both its standard input and standard output. Moreover, the `"init"` phase is very costly and must be performed once and for all. Therefore, this code has to be *locally initialized*, and needs also to have the ability to be recalled, by having its I/O channels stored.

The implementation of non-symmetric Krylov method, e.g. BiCGStab, only requires a matrix-free matrix-vector product routine aa to compute the action of matrix A on any vector λ. The `parfun` skeleton will allow us to make this *ordinary routine* be a parallel computation network.

4.2. The coupling algorithm

Then, the coupling algorithm is the following.

Initialization

- define aa that applies in parallel matrix A given by (7) to any vector $\lambda \in \Lambda$.
- compute in parallel the inverse of the matrices L_i of the inner linear systems for all subdomains and the right-hand side b given by (7).
- choose $\lambda^0 = 0$.

Iteration

- run BiCGStab with the parallel matrix-vector product aa, the right-hand side b and the initial guess λ^0.
- name the solution $\lambda^\star$.

Finalization

- solve in parallel the inner linear systems associated with $S_q\lambda^\star$ and store the $v_i^\star$'s for all subdomains.

4.3. The OCamlP3l coupling code

The coordination code itself is so short that it can be presented in extenso. The parallel annotations, i.e. the OCamlP3l skeletons, are underlined.

```
   let solve_on_all_subdomains =
     let prog () =
       let cin = ref None and cout = ref None in
       (fun (i, task, li) ->
 5        let command = Dd.make_solve_on_a_subdomain_command i in
          let ic, oc = My_unix.spawn command cin cout in
          Dd.send_task oc task;
          Dd.send_boundary_values oc li;
          flush oc;
10        Dd.receive_boundary_values ic) in
     parfun (fun () -> mapvector (seq prog, Dd.number_of_processors));;

   pardo (fun () ->
     (* Initialization *)
15   let n = Dd.number_of_subdomains in
     let f = fun l ->
       Dd.permutation (Dd.projection (solve_on_all_subdomains l)) in
     let aa = fun l -> Dd.axpy (-1.) (f (Dd.loop_vector l)) l in
     let b = f (Dd.init_vector_of_size n) in
20   let l_0 = Dd.zero_vector_of_size n in
     let algorithm = Bicgstab.algorithm Dd.axpy Dd.dot in
     (* Iteration *)
     let l_star = algorithm aa b l_0 in
     (* Finalization *)
25   solve_on_all_subdomains (Dd.final_vector l_star));;
```

The Dd module is dedicated to domain decomposition. In particular, it delivers types for the unknown vector λ that ease the implementation of the operators R_i and $\tilde{R}_i^T$.

The function `solve_on_all_subdomains` (lines 1–11) correspond to the code to be coupled of section 4.1. It is defined defined through an encapsulation of a `mapvector` skeleton inside a `parfun` skeleton to allows parallel computation of the function over all components of a vector anywhere in the sequential code. It uses local initialization to reserve memory slots to store the I/O channels.

The domain decomposition algorithm is implemented inside the `pardo` scope delimiter (lines 13–25). It uses repeatedly the previously defined stream processing network. The function `f` (lines 16–17) defines the function $f = sPS_q$ for which we are searching the fixed point. It is used in both the body of the matrix-vector product `aa` (line 18) and the computation of the right-hand side `b` (line 19).

5. Numerical Tests

Numerical tests have been run on the Cristal Cluster deployed at INRIA-Rocquencourt. It is made of sixteen 2.8 GHz Intel Xeon bi-processors with 2 Gb of RAM each interconnected on a dedicated

Gigabit network. More details can be found in [4].

5.1. Extensibility results

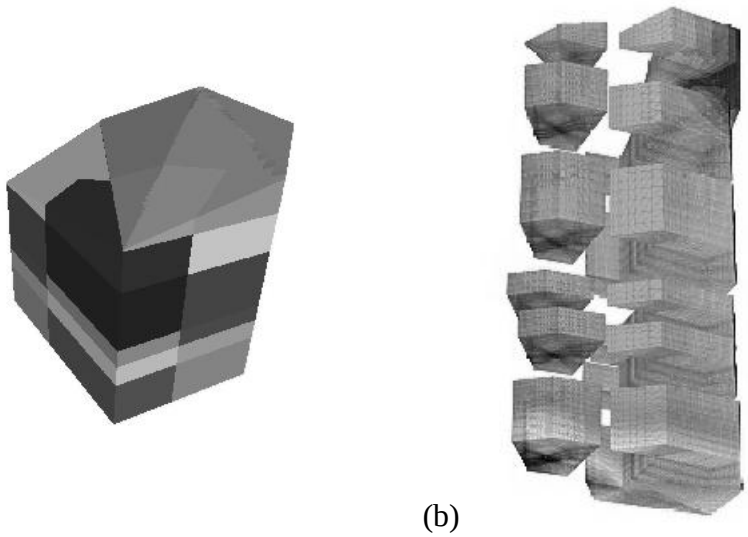
We associated one processor to each subdomain, and we increased the number of processors while keeping a constant load per processor. All subdomains were made of about 50000 cells, in order to avoid swapping problems during the initialization phase of the local subproblems. The efficiency of the coupling algorithm is related to the way the global domain is decomposed, but this always led to a flat 30%-overhead in the conforming case, and no more than twice the time in the nonconforming case, which is not bad for up to 16 subdomains, i.e. up to 800000 cells.

The main drawback of the current version of OCamlP3l is the existence of a bottleneck since all communications are centralized in the `mapvector` skeleton. Obviously, this point was not crucial in our tests, but this may become an issue for larger tests. So a specific skeleton accounting for direct communication driven by a connectivity table is under development.

5.2. A realistic 3D flow simulation

We consider now the problem of a 3D flow simulation in a realistic porous media designed by ANDRA (the French National Agency for Radioactive Waste Management) to study the feasibility of an underground nuclear waste disposal, see [5]. We have simplified the model by making all the subdomains homogeneous, but the numerical difficulty remains the flat aspect of the domain, and the high heterogeneity between the subdomains as the permeability jumps by a factor of several orders of magnitude from a geological layer to the other.

The domain is defined by a global mesh provided by ANDRA containing about 400000 hexahedric cells. It is decomposed following the geology into 12 subdomains with matching meshes at the interfaces, see Figure 2a where the unit on the vertical axis is multiplied by a factor of 100 (the domain is actually flat). The pressure solution is displayed on Figure 2b.



(a) (b)

Figure 2. A realistic 3D flow simulation. (a) The domain. (b) A split view of the pressure.

The main difficulty here was to fine tune the Robin coefficients on each interface to insure convergence of the coupling algorithm. Our empirical tests based on the physics of the porous medium

and the geometry of the domain led us to take into account the aspect ratio of the subdomains and the permeability jumps by choosing

$$\alpha_{ij} = \alpha_{ji} = \frac{2K_iK_j}{(K_i + K_j)L_{ij}^{\Omega}} \quad i,j \in I \tag{9}$$

where L_{ij}^{Ω} is the characteristic length of the domain Ω along the normal direction to the interface Σ_{ij}.

6. Conclusions

We have presented a non-overlapping domain decomposition method for non-matching meshes based on Robin interface conditions for the computation of 3D flow in porous media.

The main contribution of this paper is the way this algorithm is implemented. We have used the OCamlP3l parallel programming system that comes from the functional programming world. This system provides a structured approach to parallel programming obtained from skeletons and template based compilation techniques. The user describe its parallel algorithm by combining basic building blocks, then the (same) source code can be compiled either for sequential execution or for parallel execution, and both modes should always produce the same result. In short, this means that the user has never to take care of bugs specific to parallelism. 3D numerical results have shown the interest of the approach.

We are now working on both numerical and programming aspects: on one side, we are implementing another domain decomposition method based on Neumann-Neumann preconditioning with balancing, and on the other side, we are developing a new `mapvector` skeleton with communication capabilities based on a connectivity table that will reduce the bottleneck problem.

7. Acknowledgments

The authors thank Martial Mancip for providing an efficient implementation for a conservative L^2-projection routine, and ANDRA, the French National Agency for Radioactive Waste Management, for providing the 3D realistic test case.

References

[1] Todd Arbogast and Ivan Yotov. A non-mortar mixed finite element method for elliptic problems on non-matching multiblock grids. *Comput. Methods Appl. Mech. Engrg.*, 149(1-4):255–265, 1997. Symposium on Advances in Computational Mechanics, Vol. 1 (Austin, TX, 1997).

[2] F. Clément, R. Di Cosmo, Z. Li, V. Martin, A. Vodicka, and P. Weis. Parallel programming with the OcamlP3l system. Applications to numerical code coupling. Rapport de Recherche 5131, Inria, Rocquencourt, France, 2004.

[3] Marco Danelutto, Roberto Di Cosmo, Xavier Leroy, and Susanna Pelagatti. Parallel functional programming with skeletons: the OCamlP3l experiment. *The ML Workshop*, 1998.

[4] V. Martin. *Simulations multidomaines des écoulements en milieu poreux*. PhD thesis, Université de Paris 9, France, 2004.

[5] G. Pépin, B. Vialay, and D. Perraud. Cahier des charges relatif à la réalisation de calculs de transport de radionucléides avec le code castem2000. Cahier des charges, Andra, Châtenay-Malabry, France, March 2001. In French.

[6] F. A. Rabhi and S. Gorlatch, editors. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer-Verlag, London, 2003.

[7] J.E. Roberts and J.-M. Thomas. Mixed and hybrid methods. In P.G. Ciarlet and J.L. Lions, editors, *Handbook of Numerical Analysis Vol.II*, pages 523–639. North Holland, Amsterdam, 1991.

Mondriaan sparse matrix partitioning for attacking cryptosystems by a parallel block Lanczos algorithm — a case study

Rob H. Bisseling[a], Ildikó Flesch[b]

[a]Department of Mathematics, Utrecht University, P.O. Box 80010, 3508 TA Utrecht, The Netherlands (Rob.Bisseling@math.uu.nl)

[b]Department of Information and Knowledge Systems, Institute for Computing and Information Sciences, Radboud University Nijmegen, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands (ildiko@cs.ru.nl)

Abstract. A case study is presented demonstrating the application of the Mondriaan package for sparse matrix partitioning to the field of cryptology. An important step in an integer factorisation attack on the RSA public-key cryptosystem is the solution of a large sparse linear system with 0/1 coefficients, which can be done by the block Lanczos algorithm proposed by Montgomery. We parallelise this algorithm using Mondriaan partitioning and discuss the high-level components needed. A speedup of 10 is obtained on 16 processors of a Silicon Graphics Origin 3800 for the factorisation of an integer with 82 decimal digits, and a speedup of 7 for 98 decimal digits.

1. Introduction

The security of the widely used RSA public-key cryptosystem [12] is based on the fact that finding the prime factors of a large integer is extremely time-consuming. The state-of-the-art in integer factorisation methods tells us how large the keys used in RSA must be to withstand attacks based on trying to find the prime factors for a given public-key value.

On May 9, 2005, Bahr, Böhm, Franke, and Kleinjung [1] announced a new record factorisation: with help of te Riele and Montgomery they factorised the 200 decimal-digit number originally posed as the RSA-200 challenge in 1991. This factorisation used the Number Field Sieve (NFS) [9], which is currently the best factorisation method for large integers. In practice, the NFS almost always finds a non-trivial factor of a composite number within a few attempts. The two most time-consuming parts of this method are the *sieving step* and the *matrix step*. The sieving step took from December 2003 to December 2004, and was done by farming out jobs to a variety of computers, taking a total of 55 CPU years (at the equivalent speed of a 2.2 GHz Opteron processor). The matrix step is more tightly coupled and needs more memory since it involves a large sparse matrix, with 64 million rows and columns and 11×10^9 nonzeros for RSA-200. Because of this, it must be carried out on a parallel computer. The matrix step took about three months on a cluster of 80 Opterons. The linear system of the matrix step was solved by a block Wiedemann algorithm [5]. An alternative method would be the block Lanczos algorithm proposed by Montgomery [10].

In the present work, we will discuss the high-level components needed in a parallel computation of the matrix step, such as Mondriaan matrix partitioning. In this paper, we focus on the block Lanczos algorithm, but the same high-level components are also needed for the block Wiedemann algorithm.

2. Sequential algorithm

2.1. Construction of the sparse matrix

The sieving step in the factorisation of a large number n tries to find many pairs (a_j, b_j) of integers such that $a_j \equiv b_j \pmod{n}$ and a_j and b_j are the product of squares and small primes. Let p_i be the ith prime, i.e., $p_1 = 2, p_2 = 3, p_3 = 5$, etc. and let $p_0 = -1$. Then we can write each a_j uniquely as a finite product

$$a_j = \prod_i p_i^{m_{ij}}. \tag{1}$$

Note that a_j is square if and only if all exponents m_{ij} are even. Define a matrix A by $a_{ij} = m_{ij} \bmod 2$. The matrix A is sparse because integers have only a limited number of prime factors. Define a similar matrix B for the integers b_j.

The matrix step tries to construct a subset of pairs $(a_j, b_j),\ j \in S$, such that $\prod_{j\in S} a_j$ and $\prod_{j\in S} b_j$ are both square. Let $\alpha^2 = \prod_{j\in S} a_j$ and $\beta^2 = \prod_{j\in S} b_j$. If $\gcd(\alpha\beta, n) = 1$, then $\gcd(\alpha - \beta, n)$ is a factor of n, and hopefully a non-trivial one. If we write $S = \{j :\ x_j = 1\}$, where $\mathbf{x}$ is an integer vector with 0/1 components x_j, we see that the two products are square if and only if $A\mathbf{x} = 0$ and $B\mathbf{x} = 0$, where all computations are carried out modulo 2, i.e., in the finite field GF(2). Let the $n_1 \times n_2$ matrix C represent the two simultaneous linear systems,

$$C = \begin{bmatrix} A \\ B \end{bmatrix}.$$

Thus, we need to solve $C\mathbf{x} = 0$. Figure 1 shows an example matrix C.

2.2. Sequential block Lanczos algorithm

To find (part of) the nullspace $\mathcal{N}(C)$ of C, we can apply the block Lanczos algorithm as proposed by Montgomery [10]. Since this algorithm is only suitable for symmetric matrices, it is applied to C^TC in such a way that it finds a nullspace $\mathcal{N}(C^TC)$ that is as large as possible. There is no need to form the product C^TC explicitly: multiplication by C^TC is carried out as multiplication by C followed by multiplication by C^T. Furthermore, it suffices to store only C. Since $\mathcal{N}(C) \subset \mathcal{N}(C^TC)$, we hope to be able to find some vector $\mathbf{x} \in \mathcal{N}(C)$; this can be done by a postprocessing procedure [10] after the block Lanczos algorithm.

The block Lanczos algorithm is applied to solve $C^TCX = C^TCY$, where X and Y are $n_2 \times N$ matrices. The solution matrix X contains a set of N columns, each representing a solution $\mathbf{x}$. This way, we obtain N solutions in one run of the algorithm. The size of N is chosen as the word size of an integer on the computer used, say $N = 32$. This choice enables storage of the complete matrix X in a single one-dimensional integer array of length n_2. It also allows the use of efficient bit operations in the algorithm. The matrix Y is chosen as a random bit matrix. The aim of this approach is to obtain many independent solutions of $C^TC\mathbf{x} = 0$, given by the columns of the matrix $X - Y$.

Algorithm 1 summarises the steps of the block Lanczos algorithm. At the first occurrence of a matrix in the algorithm, its size is given as a superscript. Matrix subscripts denote the order in a sequence of matrices. All multiplication operations are explicitly shown by using an asterisk, e.g. $C * V$ on line 15. The result matrix is then written as CV. There is no need to store both V and V^T; only one of the two matrices suffices. The matrix $Cond$ represents the termination condition; SS^T represents the index set S. The function generateWS generates new matrices W^{inv} and SS^T. It only involves computations on small $N \times N$ matrices. Its description is omitted for the sake of brevity; see [10, Fig. 1] for details.

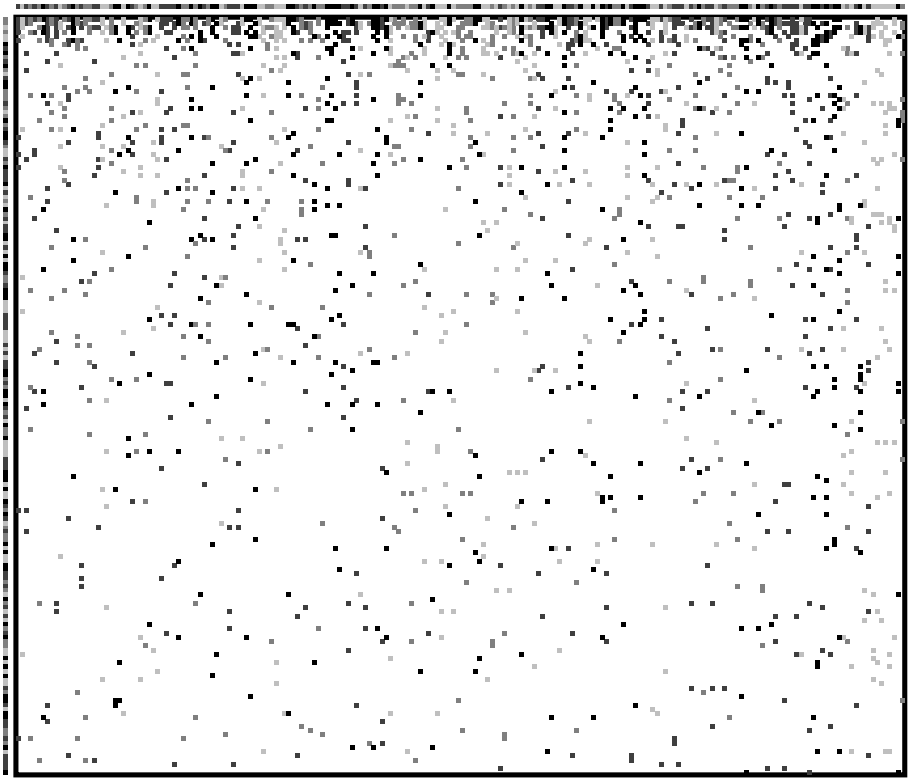


Figure 1. Matrix C corresponding to a sparse linear system $C\mathbf{x} = 0 \pmod 2$ obtained by the Multi-Polynomial Quadratic Sieve (MPQS) method for a 30-decimal integer. The 179×210 matrix C has 1916 nonzero elements. Each matrix row represents a prime; each column a pair of integers (a_j, b_j). The primes are sorted in increasing order, where each prime may occur at most twice. The matrix has been partitioned for four processors of a parallel computer, shown in different grey shades, by using the Mondriaan package [14]. Also shown is a partitioning of the input vector (above the matrix) and output vector (left) for a sparse matrix–vector multiplication $\mathbf{y} = C\mathbf{x}$. Matrix source: courtesy of Richard Brent.

3. High-level components for parallel computation

The matrix step of the block Lanczos algorithm requires the following major high-level components:

- sparse matrix–vector multiplication;
- sparse matrix partitioning;
- vector partitioning;
- dense vector inner-product computation;
- AXPY operation;
- global-local indexing mechanism.

Algorithm 1 The sequential block Lanczos algorithm.

Input: matrices C of size $n_1 \times n_2$ and Y of size $n_2 \times N$.
Output: matrix X, such that $C^T(CX) = C^T(CY)$.
1. Initialise:
1a. $W_{-2}^{\text{inv}}{}^{N \times N} = W_{-1}^{\text{inv}}{}^{N \times N} = 0$
1b. ${V_{-2}}^{n_2 \times N} = {V_{-1}}^{n_2 \times N} = 0$
1c. ${CV_{-1}}^{n_1 \times N} = 0$
1d. ${K_{-1}}^{N \times N} = 0$
1e. ${SS^T}_{-1}^{N \times N} = I_N$
1f. $X^{n_2 \times N} = 0$
2. ${V_0}^{n_2 \times N} = C^T * (C * Y)$
3. ${CV_0}^{n_1 \times N} = C * V_0$
4. ${Cond_0}^{N \times N} = (CV_0)^T * CV_0$
5. $i = 0$
while $Cond_i \neq 0$ **do**
7. $[W_i^{\text{inv}}, SS_i^T] = \text{generateWS}(Cond_i, SS_{i-1}^T, N, i)$
8. $X = X + V_i * (W_i^{\text{inv}} * (V_i^T * V_0))$
9. $C^T CV_i^{n_2 \times N} = C^T * CV_i$
10. $K_i = ((CV_i)^T * (C * (C^T CV_i))) * SS_i^T + Cond_i$
11. $D_{i+1}^{N \times N} = I_N - W_i^{\text{inv}} * K_i$
12. $E_{i+1}^{N \times N} = -W_{i-1}^{\text{inv}} * (Cond_i * SS_i^T)$
13. $F_{i+1}^{N \times N} = -W_{i-2}^{\text{inv}} * (I_N - Cond_{i-1} * W_{i-1}^{\text{inv}}) * K_{i-1} * SS_i^T$
14. $V_{i+1} = C^T CV_i * SS_i^T + V_i * D_{i+1} + V_{i-1} * E_{i+1} + V_{i-2} * F_{i+1}$
15. $CV_{i+1} = C * V_{i+1}$
16. $Cond_{i+1} = (CV_{i+1})^T * CV_{i+1}$
17. $i = i + 1$
18. Return X.

These components can be viewed as building blocks that occur in many different applications; for instance, they occur in both the block Lanczos algorithm and the block Wiedemann algorithm for the matrix step, but also in most iterative linear system solvers.

Sparse matrix–vector multiplication involving the matrix C occurs on lines 2, 3, 9, 10, 15 of Algorithm 1. Here, the sparse bit matrix C is multiplied by an $n_2 \times N$ bit matrix, which can be viewed as N multiplications by a bit vector of length n_2, or the transpose matrix C^T is multiplied. The parallel component required is a four-phase algorithm, consisting of: (i) communication of the components of the input vectors to exactly those processors that need them; (ii) local matrix–vector multiplication; (iii) communication of local results to the owner of the corresponding output vector component; (iv) and finally addition of these results. For more details, see [2, Chap. 4]. This parallel algorithm for sparse matrix–vector multiplication improves upon that used by Montgomery [11] in his parallel version of the block Lanczos algorithm because in our approach the communication exploits the sparsity of the matrix C; in [11], however, the amount of communication is as large as for a dense matrix. The algorithm should work for every possible distribution of the matrix and the input and output vectors. Note that the algorithm is a generalisation of the regular sparse matrix–vector multiplication to the multi-vector case.

Sparse matrix partitioning can be done by any of the available sparse matrix partitioners based on multilevel hypergraph partitioners, such as hMetis [7], Mondriaan [14], Par$k$way [13], PaToH [4], or Zoltan [6]. Par$k$way and Zoltan are able to do the partitioning itself in parallel. In the present work, we used the sequential partitioner Mondriaan.

Vector partitioning can be done by algorithms that try to balance the communication load of the sparse matrix–vector multiplication. Such a partitioning is incorporated in the Mondriaan package, and an improved version is described in [3]. Note that in the block Lanczos algorithm we have two types of vectors: those of length n_1 and those of length n_2. The two types can be partitioned independently, taking the result of the preceding matrix partitioning into account.

Dense vector inner-product computation occurs on lines 4, 8, 10, 16. It is easiest to perform if all vectors of the same length have the same distribution. The vector partitioning in Mondriaan does not take the number of components assigned to each processor into account, although in practice this number is not too badly balanced among the processors. A possible extension would be to perform the vector partitioning for multiple objectives, including balancing the inner-product computation.

The *AXPY operation* ('A times X Plus Y') is a well-known level 1 operation [8] from the Basic Linear Algebra Subprograms (BLAS). In iterative linear system solvers, it has the form $\mathbf{y} := \alpha\mathbf{x}+\mathbf{y}$, where $\mathbf{x}$ and $\mathbf{y}$ are vectors and α is a scalar. Its double-precision version is sometimes called DAXPY. In Algorithm 1, the AXPY operation occurs on lines 8 and 14. For instance, on line 8, we can view the multiplication of the $n_2 \times N$ bit matrix V_i by an $N \times N$ matrix as the multiplication of an integer vector of length n_2 by a small object, analogous to a scalar. An AXPY is carried out in parallel by replicating the scalar so that every processor has a copy and letting each processor multiply the local part of the vector by the scalar.

A *global-local indexing mechanism* is needed at the start of the block Lanczos algorithm. After the matrix and vectors have been distributed, the processors know which matrix elements and vector components they own, but they do not know where to obtain the input vector components they need in the matrix–vector multiplication, or where to send contributions for output vectors. The solution to this problem is that the global address of every vector component is first stored at a location that can be inspected by all processors. This location is called the *notice board* in BSPedupack [2], or the *data directory* in Zoltan [6], which provides many additional services besides partitioning. For example, the address of component x_j with global index j of a vector $\mathbf{x}$ can be found at processor $j \bmod p$ at local index $\lfloor j/p \rfloor$, where p is the number of processors. After the address has been retrieved at the start of the block Lanczos algorithm, the current value of vector component x_j can be obtained from that address each time it is needed.

All other parts of Algorithm 1 are less important. Matrices of size $N \times N$ are small (only N integers) and can easily be communicated and replicated. For instance, the computations of lines 7, 11, 12, 13 are carried out redundantly by every processor.

4. Numerical experiments

We performed numerical experiments on up to 16 processors of the Silicon Graphics Origin 3800 at SARA in Amsterdam. We used two matrices, `c82` and `c98a`, produced by the MPQS method during the factorisation of composite integers with 82 and 98 decimal digits, respectively. For problems of this size, MPQS is faster than NFS. The matrices originating in MPQS are similar to those of NFS and representative of the wider class of sieving matrices. The properties of these matrices are given in Table 1.

We implemented the parallel block Lanczos algorithm using the high-level components described in Section 3. We partitioned the two matrices and the corresponding vectors using the Mondriaan

Name	n_1	n_2	Nonzeros
c82	16307	16338	507716
c98a	56243	56274	2075889

Table 1
The properties of the two test matrices.

package [14] version 1. The execution times of the parallel program with $p = 1$ for c82 and c98a are about 80 s and 1200 s, respectively. We only have a parallel version of the program available, so we cannot use a sequential version to compare with. Thus, there will be some overhead in our reference version, which is the parallel program run on one processor. The overhead mainly consists of global-local index transformations and unnecessary calls to the synchronisation mechanism. This overhead is expected to be small, because the index transformations are carried out in a preprocessing step, and thus are removed from the main loop of the computation. Furthermore, the main loop contains only a few synchronisations. For $p = 1$, all communications reduce to memory copies. The relative speedup of the parallel program compared to the $p = 1$ case is given in Figures 2 and 3. Note that we achieve a higher speedup on the smaller problem, which is unusual, and which we find hard to explain. Certainly, cache effects must play a role here. (We have partially optimised our implementation to make it cache friendly.) The speedups achieved are reasonable, but not optimal. One reason for this is that the vector partitioning should be improved. The current tests used the vector partitioning of Mondriaan version 1; for version 2, we expect better results.

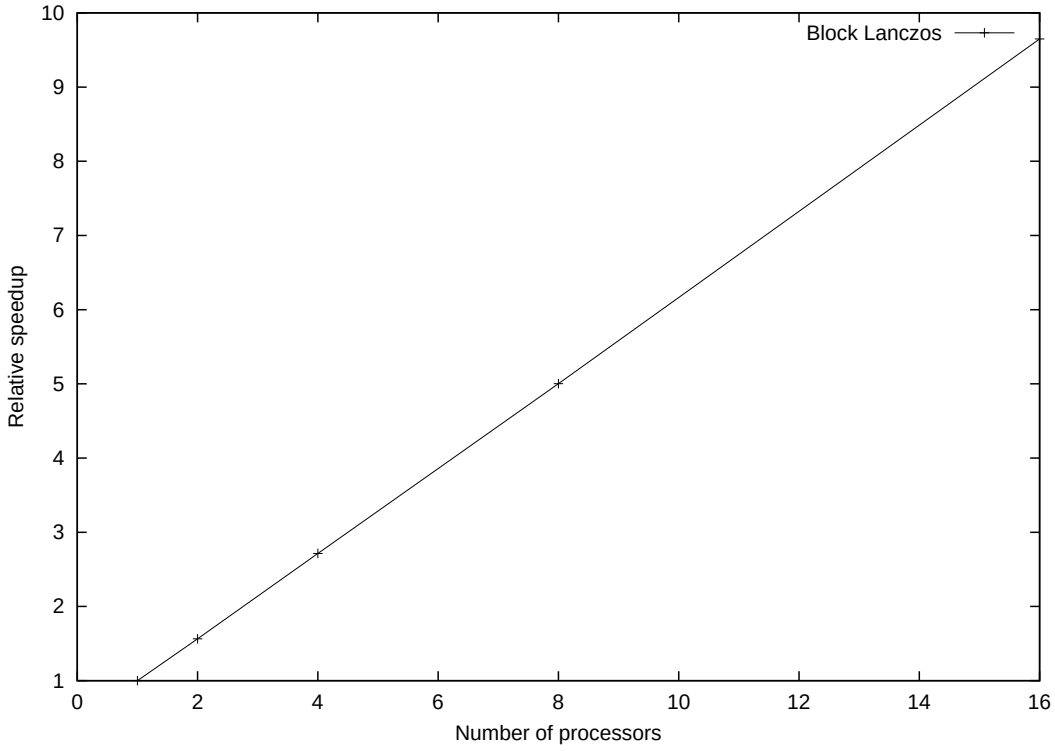


Figure 2. Speedup of parallel block Lanczos algorithm for test matrix c82.

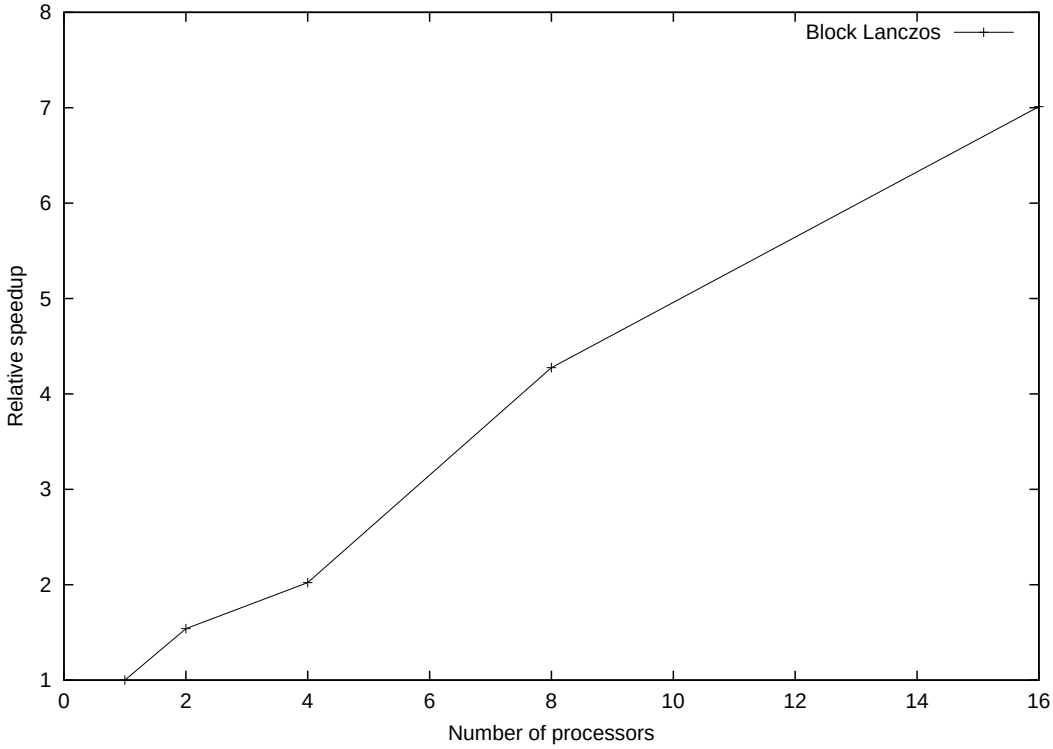


Figure 3. Speedup of parallel block Lanczos algorithm for test matrix `c98a`.

5. Conclusions and future work

We have studied an application in the field of cryptology, the solution of sparse linear systems in the binary field GF(2). We have identified important high-level components for this application and discussed their parallel aspects. This application has some particular characteristics (e.g. the computations modulo 2), but otherwise it stands for a much larger class of applications such as iterative methods for the solution of linear systems and eigensystems. The identified high-level components are important for this whole class.

An issue that also emerged from this application is that we cannot balance the computational load completely by preprocessing to find good matrix and vector partitionings. If we make use of the current bit pattern in the vectors and in the small $N \times N$ matrices (with $N = 32$) to avoid certain unnecessary operations, we save work but we also introduce a dependence of the work load on the current state. This may lead to load imbalance. It is a challenge to find a dynamic procedure to mitigate this effect.

Much research is carried out these days on higher-level tools for parallelisation. The high-level components identified here could provide focus for these efforts. If the tools would help in developing efficient and flexible components for the block Lanczos algorithm, this would have an impact on a wide range of applications.

Acknowledgements

We thank Richard Brent for providing the test matrices used in this paper and for many valuable suggestions. We thank Fatima Abu Salem for interesting and helpful discussions on integer factorisation. We also thank Herman te Riele for useful comments on the initial version of this paper. Part of the research has been funded by the Dutch BSIK/BRICKS MSV1-2 project. Computer time has been partially funded by the Dutch National Computer Facilities foundation (NCF).

References

[1] Friedrich Bahr, M. Böhm, Jens Franke, and Thorsten Kleinjung. Factorisation of RSA-200. Announcement, http://www.loria.fr/ zimmerma/records/rsa200, May 9, 2005.

[2] Rob H. Bisseling. *Parallel Scientific Computation: A Structured Approach using BSP and MPI*. Oxford University Press, Oxford, UK, March 2004.

[3] Rob H. Bisseling and Wouter Meesen. Communication balancing in parallel sparse matrix-vector multiplication. *Electronic Transactions on Numerical Analysis*, 21:47–65, 2005. Special Issue on Combinatorial Scientific Computing.

[4] Ümit V. Çatalyürek and Cevdet Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, 1999.

[5] Don Coppersmith. Solving homogeneous linear equations over GF(2) via block Wiedemann algorithm. *Mathematics of Computation*, 62(205):333–350, 1994.

[6] Karen Devine, Erik Boman, Robert Heaphy, Bruce Hendrickson, and Courtenay Vaughan. Zoltan data management services for parallel dynamic applications. *Computing in Science and Engineering*, 4(2):90–97, March/April 2002.

[7] George Karypis and Vipin Kumar. Multilevel k-way hypergraph partitioning. In *Proceedings 36th ACM/IEEE Conference on Design Automation*, pages 343–348. ACM Press, New York, 1999.

[8] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, 1979.

[9] A. K. Lenstra, H. W. Lenstra, Jr., M. S. Manasse, and J. M. Pollard. The number field sieve. In *Proceedings 22nd Annual ACM Symposium on the Theory of Computation*, pages 564–572, 1990.

[10] Peter L. Montgomery. A block Lanczos algorithm for finding dependencies over GF(2). In *Proceedings EUROCRYPT'95*, volume 921 of *Lecture Notes in Computer Science*, pages 151–168. Springer-Verlag, Berlin, 1995.

[11] Peter L. Montgomery. Parallel block Lanczos. In *Proceedings RSA-2000*, 2000.

[12] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[13] Aleksandar Trifunovic and William J. Knottenbelt. A parallel algorithm for multilevel k-way hypergraph partitioning. In *Proceedings Third International Symposium on Parallel and Distributed Computing, Cork, Ireland*, July 2004.

[14] Brendan Vastenhouw and Rob H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Review*, 47(1):67–95, 2005.

Efficient representation and parallel computation of string-substring longest common subsequences

Alexander Tiskin[a]

[a]Department of Computer Science, The University of Warwick, Coventry CV4 7AL, UK

Given two strings a, b of length m, n respectively, the string-substring longest common subsequence (SS-LCS) problem consists in computing the length of the longest common subsequence of a and every substring of b. An explicit representation of the output lengths is of size $\Theta(n^2)$. We show that the output can be represented implicitly by a set of n two-dimensional integer points, where individual output lengths are obtained by dominance counting queries. This leads to a data structure of size $O(n)$, which allows to query an individual output length in time $O(\frac{\log n}{\log\log n})$, using a recent result by JaJa, Mortensen and Shi. The currently best sequential SS-LCS algorithm by Alves et al. can be adapted to produce the output in the above geometric representation. We also develop a new parallel SS-LCS algorithm that runs on a p-processor coarse-grained computer in $O(\frac{mn}{p})$ local computation, $O(n \log p)$ communication, $O(\log p)$ barrier synchronisations, and $O(n)$ memory per processor, producing the output in the above geometric representation. Compared to previously known results, our approach presents a substantial improvement in algorithm functionality, output representation efficiency, communication efficiency and/or memory efficiency.

1. Introduction

In this paper, we consider the string-substring longest common subsequence (SS-LCS) problem, an important special case of the local sequence alignment problem, which has numerous applications in computational biology (see e.g. [7, Chapter 6], as well as references in [2,3]). Given two strings a, b of lengths m, n respectively, the SS-LCS problem consists in computing the length of the longest common subsequence of a and every substring of b. If the output lengths are represented explicitly, the total size of the output is $\Theta(n^2)$. To reduce the storage requirements, we allow the output lengths to be represented implicitly by a smaller data structure that allows efficient retrieval of individual output values. It is well-known [8,2,3] that a solution to the SS-LCS problem can be represented by a data structure of size $O(n)$. Retrieval of an individual output length typically requires scanning of at least a constant fraction of this data structure, and therefore takes time $O(n)$. In this paper, we show that the output lengths can be represented by a set of n two-dimensional integer points, where individual output lengths are obtained by dominance counting queries. This leads to a data structure of size $O(n)$, that allows to query an individual output length in time $O(\frac{\log n}{\log\log n})$, using a recent result by JaJa, Mortensen and Shi [6]. The described approach presents a substantial improvement in SS-LCS query efficiency over previous approaches.

Alves et al. [3] proposed a sequential SS-LCS algorithm, based on an idea of Schmidt [9], that runs in $O(mn)$ time and $O(n)$ memory, obtaining an implicit representation of the output. This algorithm can be adapted to produce the output in our more efficient geometric representation, without any increase in asymptotic time or memory requirements.

The SS-LCS problem can be solved in the more general setting of computing all boundary-to-boundary longest (or shortest) paths in a weighted grid graph. The first coarse-grained parallel

algorithm for this more general problem was proposed by Alves et al. [1]. The algorithm runs on a p-processor coarse-grained computer in $O(\frac{n^2 \log m}{p})$ local computation, $O(\frac{nm \log p}{p})$ communication, $O(\log p)$ barrier synchronisations, and $O(\frac{nm}{p})$ memory per processor, obtaining the output path lengths explicitly. For the SS-LCS problem proper, this was improved upon by Alves et al. [2], based on ideas of Lu and Lin [8]. Their algorithm runs in $O(\frac{mn}{p})$ local computation, $O(m^{1/2} n \log p)$ communication, $O(\log p)$ barrier synchronisations, and $O(m^{1/2} n)$ memory per processor, obtaining an implicit representation of the output. In this paper, we propose a parallel algorithm that runs in $O(\frac{mn}{p})$ local computation, $O(n \log p)$ communication, $O(\log p)$ barrier synchronisations, and $O(n)$ memory per processor, producing the output in our geometric representation. This is a substantial improvement over [2] in communication and memory efficiency, as well as the output query efficiency.

2. Problem statement and notation

Given a string, we distinguish between its contiguous *substrings*, and not necessarily contiguous *subsequences*. For two strings $a = a_1 a_2 \ldots a_m$ and $b = b_1 b_2 \ldots b_n$ of lengths m, n respectively, the *string-substring longest common subsequence (SS-LCS) problem* consists in computing the length of the longest common subsequence of a and every substring of b.

In addition to standard (non-negative) integer indices $0, 1, 2, \ldots$, we use (non-negative) *odd half-integer*[1] indices $\frac{1}{2}, \frac{3}{2}, \frac{5}{2}, \ldots$. For two integers i, j, we write $i \trianglelefteq j$ if $j - i \in \{0, 1\}$, and $i \triangleleft j$ if $j - i = 1$. We denote

$$[i : j] = \{i, i+1, \ldots, j-1, j\} \qquad \langle i : j \rangle = \left\{i + \tfrac{1}{2}, i + \tfrac{3}{2}, \ldots, j - \tfrac{3}{2}, j - \tfrac{1}{2}\right\}$$

For a function f and a predicate P defined on a variable i, we write $\mathrm{any}_{i:P(i)}\, f(i)$ to denote an arbitrary element of the set $\{f(i) : P(i)\}$. This is analogous to the use of $\min_{i:P(i)} f(i)$ to denote the minimum element of this set[2].

3. Problem analysis

It is well-known that an instance of the SS-LCS problem can be represented by a weighted *grid dag (directed acyclic graph)*, defined on a set of nodes $v_{l,i}$, where $l \in [0 : m]$, $i \in [0 : n]$. The edges are defined as follows:

edge $v_{l,i-1} \to v_{l,i}$ of weight 0, for all $l \in [0 : m]$, $i \in [1 : n]$;
edge $v_{l-1,i} \to v_{l,i}$ of weight 0, for all $l \in [1 : m]$, $i \in [0 : n]$;
edge $v_{l-1,i-1} \to v_{l,i}$ of weight 1 if $a_l = b_i$, for all $l \in [1 : m]$, $i \in [1 : n]$.

A common subsequence of string a and substring $b_{i+1} \ldots b_j$ of length r corresponds to a path $v_{0,i} \rightsquigarrow v_{m,j}$ of total weight r. The solution to the SS-LCS problem is equivalent to finding the weight $A(i, j)$ of a longest (heaviest) path $v_{0,i} \rightsquigarrow v_{m,j}$ for all $i, j \in [0 : n]$. If $i = j$, we have $A(i, j) = 0$. By convention, if $j < i$, then we let $A(i, j) = j - i$.

[1] It would be possible to reformulate all our results using only integers. However, using half-integers helps to make the exposition simpler and more elegant.

[2] In fact, "min" (or "max") can always be used instead of "any"; however, such usage would be somewhat misleading when "any" happens to be sufficient.

Theorem 1. *Values $A(i,j)$ have the following properties:*

$$A(i,j) \trianglelefteq A(i-1,j); \tag{1}$$
$$A(i,j) \trianglelefteq A(i,j+1); \tag{2}$$
$$\text{if } A(i,j+1) \triangleleft A(i-1,j+1), \text{ then } A(i,j) \triangleleft A(i-1,j); \tag{3}$$
$$\text{if } A(i-1,j) \triangleleft A(i-1,j+1), \text{ then } A(i,j) \triangleleft A(i,j+1). \tag{4}$$

Proof. A path $v_{0,i-1} \rightsquigarrow v_{m,j}$ can be obtained by $v_{0,i-1} \rightarrow v_{0,i} \rightsquigarrow v_{m,j}$. Therefore, $A(i,j) \leq A(i-1,j)$. On the other hand, any path $v_{0,i-1} \rightsquigarrow v_{m,j}$ consists of a subpath $v_{0,i-1} \rightsquigarrow v_{l,i}$ of weight at most 1, followed by a subpath $v_{l,i} \rightsquigarrow v_{m,j}$. Therefore, $A(i,j) \geq A(i-1,j) - 1$. We thus have (1) and, by symmetry, (2).

A crossing pair of paths $v_{0,i} \rightsquigarrow v_{m,j}$ and $v_{0,i-1} \rightsquigarrow v_{m,j+1}$ can be rearranged into a non-crossing pair of paths $v_{0,i-1} \rightsquigarrow v_{m,j}$ and $v_{0,i} \rightsquigarrow v_{m,j+1}$. Therefore, we have *the Monge property*:

$$A(i,j) + A(i-1,j+1) \leq A(i-1,j) + A(i,j+1)$$

Rearranging the terms

$$A(i-1,j+1) - A(i,j+1) \leq A(i-1,j) - A(i,j)$$

and applying (1), we obtain (3) and, by symmetry, (4). ■

The properties of Theorem 1 are symmetric with respect to i and $n-j$. Alves et al. [2,3] introduce the same properties but do not make the most of their symmetry. We aim to exploit symmetry to the full.

Corollary 1. *Values $A(i,j)$ have the following properties:*

$$\text{if } A(i,j) \triangleleft A(i-1,j), \text{ then } A(i,j') \triangleleft A(i-1,j') \text{ for all } j' \leq j;$$
$$\text{if } A(i,j) = A(i-1,j), \text{ then } A(i,j') = A(i-1,j') \text{ for all } j' \geq j.$$

Also,

$$\text{if } A(i,j) \triangleleft A(i,j+1), \text{ then } A(i',j) \triangleleft A(i',j+1) \text{ for all } i' \geq i;$$
$$\text{if } A(i,j) = A(i,j+1), \text{ then } A(i',j) = A(i',j+1) \text{ for all } i' \leq i.$$

Proof. In both pairs, the properties are each other's converse and an immediate consequence of Theorem 1. ■

Informally, Corollary 1 says that, if values A are represented as a matrix, then the inequality between the corresponding elements in two successive rows (respectively, columns) "propagates to the left (respectively, downwards)", and the equality "propagates to the right (respectively, upwards)". Recall that by convention, $A(i,j) = j - i$ for all index pairs $j < i$. Therefore, we always have an inequality between the corresponding elements in successive rows or columns in the lower triangular part of matrix A. If we fix i and scan the set of indices j from left ($j = 0$) to right ($j = n$), an inequality may change to an equality at most once. We call such a value of j *critical* for i. Symmetrically, if we fix j and scan the set of indices i from bottom ($i = n$) to top ($i = 0$), an inequality may change to an equality at most once, and we can identify values of i that are critical for j. Crucially, for all pairs (i,j), index i will be critical for j if and only if index j is critical for i. This property lies at the core of our method, which is based on the following definition.

Definition 1. *An odd half-integer point* $(i,j) \in \langle 0:n\rangle^2$ *is called* A-critical, *if*

$$A\big(i+\tfrac{1}{2}, j-\tfrac{1}{2}\big) \lhd A\big(i-\tfrac{1}{2}, j-\tfrac{1}{2}\big) = A\big(i+\tfrac{1}{2}, j+\tfrac{1}{2}\big) = A\big(i-\tfrac{1}{2}, j+\tfrac{1}{2}\big)$$

In particular, point (i,j) is never A-critical for $i > j$. Point (i,j) is A-critical for $i = j$, iff $A\big(i-\frac{1}{2}, j+\frac{1}{2}\big) = 0$.

Corollary 2. *For each* i *(respectively,* j*), there exists at most one* j *(respectively,* i*) such that the point* $(i,j) \in \langle 0:n\rangle^2$ *is* A*-critical.*

Proof. By Corollary 1 and Definition 1. ■

Definition 1 and Corollary 2 allow us to represent an $(n+1)\times(n+1)$ SS-LCS matrix by its set of critical points, which are at most n. The following theorem shows that such representation is unique, and gives a simple formula for recovering matrix elements.

Definition 2. *Point* (i_0,j_0) dominates[3] *point* (i,j), *if* $i_0 < i$ *and* $j < j_0$.

Informally, the dominated point is "below and to the left" of the dominating point.

Theorem 2. *For an arbitrary integer point* $(i_0,j_0) \in [0:n]^2$, *let* $d_A(i_0,j_0)$ *denote the number of* A*-critical points it dominates. We have*

$$A(i_0,j_0) = j_0 - i_0 - d_A(i_0,j_0)$$

Proof. Induction on $j_0 - i_0$. ■

There is a close connection between Theorem 2 and the canonical representation of general Monge matrices (see e.g. [5]). The difference is that in the special case of SS-LCS matrices, the representation size is just $O(n)$.

Informally, Theorem 2 says that the value $A(i_0,j_0)$ is determined by the number of A-critical points dominated by (i_0,j_0). Trivially, this number can be obtained by scanning the set of all critical points in time $O(n)$. Much more efficient methods exist when preprocessing of the critical point set is allowed.

The dominance relationship between two-dimensional (in general, multi-dimensional) points is a classical topic in computation geometry. The following theorems are derived from two relevant geometric results, one classical and one recent.

Theorem 3. *Given an instance of the SS-LCS problem, there exists a data structure which*
- *has size* $O(n \log n)$*;*
- *can be built from the set of critical points in time* $O(n \log n)$*;*
- *allows to query an individual output length in time* $O(\log n)$.

Proof. Use a 2D range tree [4]. The value $A(i_0,j_0)$ is then obtained by Theorem 2. ■

Theorem 4. *Given an instance of the SS-LCS problem, there exists a data structure which*
- *has size* $O(n)$*;*
- *allows to query an individual output length in time* $O(\frac{\log n}{\log\log n})$.

Proof. As in Theorem 3, but the 2D range tree is replaced by the data structure from [6]. ■

While the data structure of Theorem 4 is asymptotically more efficient, the structure of Theorem 3 is simpler, requires a less powerful computation model, and is more likely to be practical.

[3]The standard definition of dominance requires $i < i_0$ instead of $i_0 < i$. Our definition is more convenient in the context of the LCS problem.

4. The parallel algorithm

A divide-and-conquer framework for the SS-LCS problem and the more general string editing problem has been developed in [1–3]. String a is partitioned into substrings, inducing a partitioning of the representation dag into strips, such that each pair of adjacent strips shares a single row of nodes. Consider a pair of adjacent strips defined by nodes $v_{l,i}$, where $l_0 \le l \le l_1$ and $l_1 \le l \le l_2$, respectively. Denote the SS-LCS matrices in the substrips by A, B respectively. The goal is to merge these matrices to obtain the SS-LCS matrix C for the combined strip defined by nodes $v_{l,i}$, where $l_0 \le l \le l_2$.

By Theorem 2, matrices A, B, C can be represented by the sets of at most n A-, B- and C-critical points. For the special case when either of the strips is of width 1 (e.g. $l_2 - l_1 = 1$), Alves et al. [3] describe a merging procedure that runs in time $O(n)$. Their sequential algorithm based on this procedure runs in time $O(mn)$, and produces a data structure of size $O(n)$, equivalent to the set of critical points representing the solution of the original SS-LCS problem. By adding a post-processing phase based on Theorems 3, 4, the algorithm can be adapted to produce a query-efficient representation of the output.

We now present an efficient coarse-grained parallel algorithm for the SS-LCS problem. We assume the *bulk-synchronous parallel (BSP)* computation model. A *BSP computer*, introduced in [10], consists of p *processors* connected by a communication network. Each processor has a fast *local memory*. A BSP computation consists of a sequence of S *supersteps*. with costs $w_s + h_s \cdot g + l$, $1 \le s \le S$, where w_s is the superstep's local computation cost, h_s is the superstep's communication cost, and g, l are parameters of the computer. The overall cost of a BSP computation is $W + H \cdot g + S \cdot l$, where $W = \sum_{s=1}^{S} w_s$ is the total local computation cost, $H = \sum_{s=1}^{S} h_s$ is the total communication cost, and S is the total synchronisation cost.

Our parallel algorithm is based on a novel sequential strip merging procedure, which works for arbitrary strip widths $l_1 - l_0$, $l_2 - l_1$, and runs in time $O(n^{3/2})$. At the base of the recursion, we use the sequential algorithm by Alves et al. [3].

Algorithm 1. *String-substring longest common subsequences in BSP.*

Parameters: integers m, n. We assume $m \ge n^{1/2} p \log p$.

Input: strings a, b of length m and n, respectively.

Output: the set of critical points for strings a, b.

Description. The computation proceeds in two stages.

First stage. Partition string a into p substrings of length m/p, inducing a partitioning of the representation dag into strips, such that each pair of adjacent strips shares a single row of nodes. The resulting p subproblems are distributed across the processors. Each processor runs the algorithm by Alves et al. [3] on the local subproblem, obtaining a critical point set representation of the resulting SS-LCS matrix.

Second stage. Perform $\log p$ levels of pairwise subproblem merging. Each subproblem is represented by a set of at most n critical points. For two subproblems with SS-LCS matrices A, B, the sets of A- and B-critical points are collected, and the set of C-critical points is computed sequentially by a designated processor. The computation proceeds as follows.

By Theorem 2, computing C-critical points is equivalent to determining the set of values

$$d_C(i,k) = \min_{j \in [0:n]} [d_A(i,j) + d_B(j,k)]$$

for $i, k \in [0 : n]$. Assume for simplicity that n is a power of 2. We proceed by partitioning the index set $\langle 0 : n\rangle^2$ recursively into regular half-sized square blocks. For each block, we establish the number of C-critical points contained in it, and proceed with the recursive partitioning of the block as long as this number is greater than 0.

Consider an $h \times h$ block

$$\langle i_0 - h : i_0\rangle \times \langle k_0 : k_0 + h\rangle$$

The C-critical points in this block will be determined by A-critical points in $\langle i_0-h : i_0\rangle\times\langle 0 : n\rangle$, and B-critical points in $\langle 0 : n\rangle \times \langle k_0 : k_0 + h\rangle$. We call such A- and B-critical points *relevant*. For the current block, there are at most h relevant points in each of A, B.

For any $j \in [0 : n]$, let $\delta_A(j)$ (respectively, $\delta_B(j)$) denote the number of relevant A-critical (respectively, B-critical) points in $\langle i_0 - h : i_0\rangle \times \langle 0 : j\rangle$ (respectively, $\langle j : n\rangle \times \langle k_0 : k_0 + h\rangle$):

$$\delta_A(j) = d_A(i_0 - h, j) - d_A(i_0, j) \qquad \delta_B(j) = d_B(j, k_0 + h) - d_B(j, k_0)$$

Sequence δ_A is monotonically increasing from $\delta_A(0) = 0$ to $\delta_A(n) \le h$. Sequence δ_B is monotonically decreasing from $\delta_B(0) \le h$ to $\delta_B(n) = 0$.

As the block size h gets smaller, sequences δ_A, δ_B contain fewer and fewer distinct values. We represent these sequences compactly by storing, for every $d \in [-\delta_B(0) : \delta_A(n)]$, the values

$$\Delta_A(d) = \operatorname*{any}_{j:\delta_A(j)-\delta_B(j)=d} \delta_A(j) \qquad \Delta_B(d) = \operatorname*{any}_{j:\delta_A(j)-\delta_B(j)=d} \delta_B(j)$$
$$M(d) = \min_{j:\delta_A(j)-\delta_B(j)=d} [d_A(i_0, j) + d_B(j, k_0)]$$

When the set $\{j : \delta_A(j) - \delta_B(j) = d\}$ is empty, the corresponding values $\Delta_A(d)$, $\Delta_B(d)$, $M(d)$ are undefined and omitted from further computations. Sequences Δ_A, Δ_B can be computed in time $O(h)$ by a single scan of the set of relevant A- and B-critical points. Sequence M is computed in the previous recursive step by a procedure described below. From sequences Δ_A, Δ_B, M, the following values can be found in time $O(h)$:

$$d_C(i_0, k_0) = \min_{d\in[-\delta_B(0):\delta_A(n)]} M(d)$$
$$d_C(i_0 - h, k_0) = \min_{d\in[-\delta_B(0):\delta_A(n)]} [\Delta_A(d) + M(d)]$$
$$d_C(i_0, k_0 + h) = \min_{d\in[-\delta_B(0):\delta_A(n)]} [M(d) + \Delta_B(d)]$$
$$d_C(i_0 - h, k_0 + h) = \min_{d\in[-\delta_B(0):\delta_A(n)]} [\Delta_A(d) + M(d) + \Delta_B(d)]$$

The number of critical points in the current block can then be determined as

$$d_C(i_0 - h, k_0 + h) - d_C(i_0 - h, k_0) - d_C(i_0, k_0 + h) + d_C(i_0, k_0)$$

If the above value is non-zero, the recursion proceeds by partitioning the current block of size h into four subblocks of size $h/2$. The sets of relevant A- and B-critical points are split accordingly. Consider each of the four half-sized subblocks. Let i'_0, k'_0, δ'_A, δ'_B, M' denote the values defined for the subblock analogously to values i_0, k_0, δ_A, δ_B, M for the original block. For every $d \in [-\delta_B(0) : \delta_A(n)]$, let

$$\Delta^*_A(d) = \operatorname*{any}_{j:\delta_A(j)-\delta_B(j)=d} \delta'_A(j) \qquad \Delta^*_B(d) = \operatorname*{any}_{j:\delta_A(j)-\delta_B(j)=d} \delta'_B(j)$$

Similarly to Δ_A, Δ_B, sequences Δ_A^*, Δ_B^* can be computed in time $O(h)$ by a single scan of the set of relevant A- and B-critical points. For every $d' \in [-\delta'_B(0) : \delta'_A(n)]$, value $M'(d')$ can now be obtained from sequence M by

$$M'(d') = \min_{d:\Delta_A^*(d)-\Delta_B^*(d)=d'} M(d) \quad \text{for } i'_0 = i_0,\ k'_0 = k_0$$

$$M'(d') = \min_{d:\Delta_A^*(d)-\Delta_B^*(d)=d'} [\Delta_A^*(d) + M(d)] \quad \text{for } i'_0 = i_0 - \tfrac{h}{2},\ k'_0 = k_0$$

$$M'(d') = \min_{d:\Delta_A^*(d)-\Delta_B^*(d)=d'} [M(d) + \Delta_B^*(d)] \quad \text{for } i'_0 = i_0,\ k'_0 = k_0 + \tfrac{h}{2}$$

$$M'(d') = \min_{d:\Delta_A^*(d)-\Delta_B^*(d)=d'} [\Delta_A^*(d) + M(d) + \Delta_B^*(d)] \quad \text{for } i'_0 = i_0 - \tfrac{h}{2},\ k'_0 = k_0 + \tfrac{h}{2}$$

Note that evaluation of functions d_A, d_B is not required. For each of the four subblocks, every value $M(d)$ contributes to exactly one value $M'(d')$, therefore the above computation can be done in time $O(h)$.

The recursion base is $h = 1$. At this point, we establish all 1×1 blocks containing a C-critical point, which is equivalent to establishing the C-critical points themselves.

Cost analysis. *First stage.* The first stage runs in a single superstep, each processor performs $O(mn/p)$ local computation and requires $O(n)$ memory.

Second stage. The second stage runs in $\log p$ supersteps. In every superstep, we have a recursion tree of maximum degree 4, height at most $\log n$, and at most n leaves (corresponding to C-critical points).

Consider the top $\frac{\log n}{2}$ levels of the recursion tree. As we move down from the root to level $\frac{\log n}{2}$, in each level the maximum number of nodes increases by a factor of 4, and the maximum amount of computational work per node decreases by a factor of 2. Hence, the maximum amount of work per level increases in geometric progression, and is dominated by level $\frac{\log n}{2}$.

Consider the bottom $\frac{\log n}{2}$ levels of the recursion tree. Since the tree has at most n leaves, the maximum number of nodes in a level is at most n. As we move down from level $\frac{\log n}{2}$ to level $\log n$, in each level the maximum amount of computational work per node still decreases by a factor of 2. Hence, the maximum amount of work per level decreases in geometric progression, and is again dominated by level $\frac{\log n}{2}$.

Thus, the computational work in the whole recursion tree is dominated by the maximum amount of work done in level $\frac{\log n}{2}$. This level has at most n nodes, each requiring at most $O(n)/2^{\frac{\log n}{2}} = O(n^{1/2})$ work. Therefore, the local computation cost of merging two strips is at most $n \cdot O(n^{1/2}) = O(n^{3/2})$, and the overall local computation cost of the second stage is $O(n^{3/2} \log p)$.

In every superstep, the recursion tree can be evaluated depth-first. Therefore, at every given moment we are only required to store the data of the current node and its ancestors in the recursion tree. As we move down from the root to the current node, in each level the maximum amount of memory required per node decreases by a factor of 2. Hence, the overall memory required for merging two strips is dominated by the $O(n)$ memory required by the root. Therefore, the overall memory cost of the second stage is $O(n)$.

The amount of data communicated between successive supersteps is $O(n)$, therefore the communication cost of the second stage is $O(n \log p)$.

Due to the slackness assumption $m \geq n^{1/2} p \log p$, the local computation cost of the algorithm is dominated by the first stage. Overall, the algorithm runs in $O(mn/p)$ local computation, $O(n \log p)$ communication, $O(\log p)$ synchronisation, and requires $O(n)$ memory. ■

Alternatively to the second stage above, the strip representations from individual processors can be collected in a single designated processor and merged sequentially. This version of the algorithm has communication cost $O(np)$ (which is higher than that of Algorithm 1, but still an improvement over [2]), and optimal synchronisation cost $O(1)$.

As before, application of Theorems 3, 4 can significantly improve the query efficiency of our algorithm's output.

5. Conclusions

We have presented a new approach to the computation of string-substring longest common subsequences. Our approach results in a significantly improved output representation, and a coarse-grained parallel algorithm for the SS-LCS problem with improved communication and memory costs.

An immediate open question is whether the efficiency of our parallel algorithm can be improved even further, with the ultimate goal of the optimal $O\left(\frac{m+n}{p}\right)$ communication and memory, and optimal $O(1)$ synchronisation (although a solution that could achieve both of these simultaneously seems unlikely). These goals are not currently achieved even for the standard LCS problem (also known as the Levenshtein distance problem). It would also be desirable to extend the algorithm to lower values of m relative to n and p. Another interesting question is whether our algorithms can be adapted to more general sequence alignment, e.g. the general edit distance problem, or sequence alignment with non-linear gap penalties.

References

[1] C. E. R. Alves, E. N. Cáceres, F. Dehne, and S. W. Song. Parallel dynamic programming for solving the string editing problem on a CGM/BSP. In *Proceedings of the 14th ACM SPAA*, pages 275–281, 2002.

[2] C. E. R. Alves, E. N. Cáceres, and S. W. Song. A BSP/CGM algorithm for the all-substrings longest common subsequence problem. In *Proceedings of the 17th IEEE/ACM IPDPS*, pages 1–8, 2003.

[3] C. E. R. Alves, E. N. Cáceres, and S. W. Song. An all-substrings common subsequence algorithm. *Electronic Notes in Discrete Mathematics*, 19:133–139, 2005.

[4] J. L. Bentley. Multidimensional divide-and-conquer. *Communications of the ACM*, 23(4):214–229, 1980.

[5] R. E. Burkard, B. Klinz, and R. Rudolf. Perspectives of Monge properties in optimization. *Discrete Applied Mathematics*, 70:95–161, 1996.

[6] J. JaJa, C. Mortensen, and Q. Shi. Space-efficient and fast algorithms for multidimensional dominance reporting and counting. In R. Fleischer and G. Trippen, editors, *Proceedings of the 15th ISAAC*, volume 3341 of *Lecture Notes in Computer Science*, pages 558–568, 2004.

[7] N. C. Jones and P. A. Pevzner. *An introduction to bioinformatics algorithms*. Computational Molecular Biology. The MIT Press, 2004.

[8] Mi Lu and Hua Lin. Parallel algorithms for the longest common subsequence problem. *IEEE Transactions on Parallel and Distributed Systems*, 5(8):835–848, 1994.

[9] J. P. Schmidt. All highest scoring paths in weighted grid graphs and their application to finding all approximate repeats in strings. *SIAM Journal on Computing*, 27(4):972–992, 1998.

[10] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

Skeletons for Recursively Unfolding Process Topologies

Jost Berthold and Rita Loogen[a]

[a]Fachbereich Mathematik und Informatik, Philipps-Universität Marburg, Hans-Meerwein-Straße, D-35032 Marburg, Germany.

We discuss two different patterns for the generation of process topologies. All processes are either created by a single root process, or the topology unfolds recursively along a spanning tree. An obvious drawback of the first approach is the bottleneck in the root process, which becomes more serious when the number of processes increases. Difficulties of the second approach are the appropriate installation of the communication channels and the correct placement of processes on processors. We compare the generation policies for rings and toroids and analyse the impact of the topology generation on the runtimes of programs.

1. Introduction

Skeletons [3] provide commonly used patterns of parallel evaluation and simplify the development of parallel programs, because they can be used as complete building blocks in a given application context. In many skeletal parallel programming approaches a fixed number of skeletons is provided, together with optimised implementations for special target architectures, see e.g. [2] and [4] or, more recently, with implementations on top of communication libraries like MPI, see e.g. [5], [1].

In functional languages like Haskell or ML, skeletons can simply be specified as polymorphic higher-order functions. If parallelism or concurrency can be expressed, skeletons can even be *implemented* in the language itself. This is possible in languages like GpH (Glasgow parallel Haskell) [11], Concurrent Clean [8], Eden [7], or Concurrent ML [10]. Describing both the functional specification and the parallel implementation of a skeleton in the same language context has several advantages. Firstly, it constitutes a good basis for formal reasoning and correctness proofs. Secondly, it provides much flexibility, as skeleton implementations can easily be adapted to special cases, and if necessary, new skeletons can even be introduced by the programmer himself.

Topology skeletons define process systems with an underlying communication topology, e.g. pipes, rings, grids, hypercubes etc. In this paper, we discuss two different ways to generate process topologies. The simplest method is to create all processes and their interconnecting channels by a single root process. Alternatively, the topology can be unfolded recursively, each process creating its successor processes with respect to a generational spanning tree of the topology. In the first approach, the root process may become a bottleneck when the number of processes increases. Such a bottleneck is avoided in the second approach at the price of a more sophisticated installation of the communication channels and the need for explicit placement of processes on processors. In this paper, we describe and compare the two approaches for rings and toroids. We implement the skeletons in our parallel functional language Eden and briefly analyse the impact of the topology generation on the runtimes of programs.

2. A Short View on Eden

Eden [7], a parallel extension of the functional language Haskell, embeds functions into *process abstractions* with the special function `process` and explicitly *instantiates* (i.e. runs) them on remote processors using the operator `( # )`. Processes are distinguished from functions by their operational

property to be executed remotely, while their denotational meaning remains unchanged as compared to the underlying function.

```
process :: (Trans a, Trans b) => (a -> b)    -> Process a b
( # )   :: (Trans a, Trans b) => Process a b -> a -> b
```

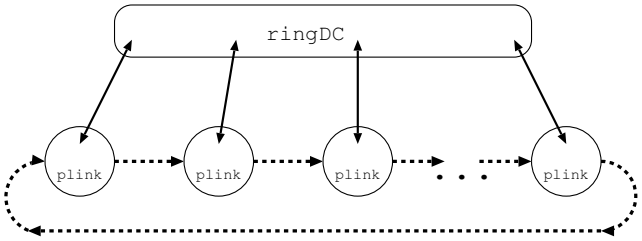
For a given function `f` and an expression `e`, evaluation of the expression `(process f # e)` leads to the creation of a new (remote) process which evaluates the function application `f e`. The argument `e` is evaluated to normal form by the creator or parent process, i.e. the process evaluating the process instantiation. The result value of `e` is transmitted from the parent to the child and the child output `f e`, which will be completely evaluated by the child process, is transmitted from the child to the parent via implicit communication channels installed during process creation. The type class `Trans` provides implicitly used functions for these transmissions. Tuples are transmitted component-wise by independent concurrent threads, and lists are transmitted as streams, element by element.

Eden provides the dynamic creation of channels which allows to establish direct channel connections between arbitrary processes. An Eden process may explicitly generate a new *dynamic reply channel* and pass the channel's name to another process. The receiving process may then either use the name to return some information directly to the sender process (*receive and use*), or pass the channel name further on to another process (*receive and pass*). Eden introduces a unary type constructor `ChanName` for the names of dynamically created channels. It provides two operators to generate and use channel names.

```
new     :: Trans a => (ChanName a -> a -> b) -> b
parfill :: Trans a =>  ChanName a -> a -> b  -> b
```

Evaluating an expression `new (\ ch_name ch_vals -> e)` has the effect that a new channel name `ch_name` is declared as reference to the new input channel via which the values `ch_vals` will eventually be received in the future. The scope of both is the body expression `e`, which is the result of the whole expression. The channel name must be sent to another process to establish the direct communication. A process can reply through a channel name `ch_name` by evaluating an expression `(parfill ch_name e1 e2)`. Before `e2` is evaluated, a new concurrent thread for the evaluation of `e1` is generated, whose normal form result is transmitted via the dynamic channel. The result of the overall expression is `e2`. The generation of the new thread is a side effect. Its execution continues independently from the evaluation of `e2`.

Figure 1 shows the definition of a ring skeleton in Eden. All processes are created by the process evaluating the function `ring` and communicate in a unidirectional way. The number of ring processes is given by the first parameter. Parameter functions `split` and `combine` specify how to distribute the input to the ring processes and how to combine the results of the ring processes to the overall result. The node function `f` determines the behaviour of each ring process. It is applied to the corresponding part of the `input` and the value received from its ring predecessor, yielding an element of the list `toParent` which is part of the overall result, and a value that is passed to its ring successor. Note that the ring is closed by using the list of ring outputs `ringOuts` rotated by one position to the right by `rightrotate` as inputs `ringIns` in the node function applications. The Haskell function `zip` converts a pair of lists element by element into a list of pairs and `unzip` does the reverse. The `mzip` function corresponds to the `zip` function except that a lazy pattern is used to match the second argument. This is necessary, because the second argument of `mzip` is the recursively defined ring input. Laziness is essential in this example - a corresponding definition is not possible in an eager language.




```
ring ::  (Trans ri,Trans ro,Trans r) =>
         Int                         -- ring size
         -> (Int -> i -> [ri])       -- input split function
         -> ([ro] -> o)              -- output combine function
         -> ((ri,r) -> (ro,r))       -- ring process mapping
         -> i -> o                   -- input-output mapping
ring n split combine f input = combine toParent
  where
    (toParent,ringOuts) = unzip [plink f # inp | inp <- nodeInputs]
    inputs              = split n input
    nodeInputs          = mzip inputs ringIns
    ringIns             = rightRotate ringOuts
    rightRotate xs      = last xs : init xs

plink :: (Trans ri,Trans ro,Trans r) =>
         ((ri,r) -> (ro,r)) -> Process (ri,ChanName r) (ro,ChanName r)
plink f = process fun_link
  where fun_link (fromParent,nextChan) =  new (\ prevChan prev ->
              let (toParent,next) = f (fromParent,prev)
              in  parfill nextChan next (toParent,prevChan))
```

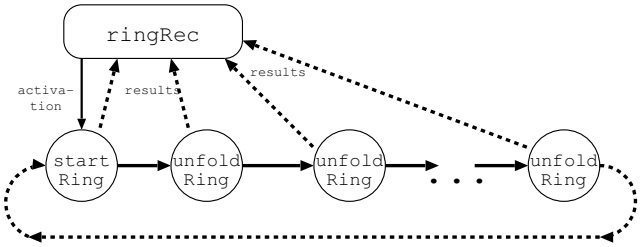
Figure 1. Eden Ring Skeleton

The function `plink` establishes direct channel connections between the ring processes. It embeds the node function `f` into a process which creates a new input channel `prevChan` that is passed to the neighbour ring process via the parent. The ring output `next` is sent via the received channel `nextChan`, while the ring input `prev` is received via its newly created input channel `prevChan`. The ring input/output from/to the parent is received and sent on static channel connections while the communication between ring processes occurs on dynamic reply channels. As all processes are created by a single parent process, the default round-robin placement policy of Eden is sufficient to guarantee an even distribution of processes on processors.

In the following section we will discuss alternative skeleton definitions where the topologies are recursively unfolded. For simplicity we restrict the discussion to rings and toroids (two-dimensional ring structures). The same techniques can be applied to higher-dimensional structures like hyper-grids or hypercubes.

3. Recursively Unfolding Rings and Toroids

The single-source creation of process systems may lead to a serious bottleneck in the creator process when the number of processes increases. For this reason, we investigate the recursive unfolding of process topologies. We start with the discussion of a one-dimensional unidirectional ring skeleton



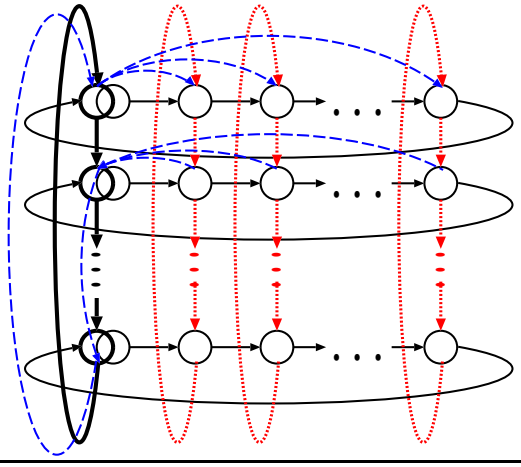

```
ringRec n split combine f input = plist 'seq' combine toParent
  where (pChans, toParent)  = createChans n  -- result channels
        plist = (process (startRing  f (split n input))) # pChans

startRing :: (Trans ri, Trans ro, Trans r) =>
             ((ri,r) -> (ro,r)) -> [ri] -> [ChanName ro]  -> ()
startRing  f (i:is) (c:cs)
   = new (\ firstChan firstIns ->  -- channel to close the ring
          let (result,ringOut)  =  f (i,firstIns)
              recCall   = unfoldRing firstChan f is
              next      = (process recCall) # (cs,ringOut)
          in  parfill c result next )

unfoldRing :: (Trans ri, Trans ro, Trans r) =>
              ChanName r -> ((ri,r) -> (ro,r)) -> [ri] ->
              ([ChanName ro],r) -> ()
unfoldRing   firstChan f (i:is) ((c:cs),ringIn)
   = parfill c result next
   where  (result, ringOut) = f (i,ringIn)
          recCall           = unfoldRing  firstChan f is
          next | null is    = parfill firstChan ringOut ()
               | otherwise  = (process recCall) # (cs,ringOut)

createChans :: Trans a => Int -> ([ChanName a],[a])
createChans 0 = ([],[])
createChans n = new (\chX valX -> let (cs,xs) = createChans (n-1)
                                  in (chX:cs,valX:xs))
```

Figure 2. Recursively Unfolding Ring Skeleton

where each process but the last creates its successor process in the ring. Subsequently, we show how higher-dimensional structures can be built on this.

3.1. Rings

Figure 2 shows an alternative definition of the ring skeleton which has the same type interface as the original one of Figure 1. The input to the ring processes is now passed as a parameter and thus will be communicated together with the process instantiation, while the output of the ring processes is returned to the originator process via initially created dynamic reply channels `pChans` which are communicated to the ring processes. The static output of the ring processes is merely the unit value (). The explicit demand on the unit value `plist` by `plist 'seq'` leads to the immediate creation



Recursive Toroid Creation
Solid lines show the underlying ring skeletons (thick lines indicate the first column ring). Dotted lines indicate the vertical connections created using dynamic channels.

Dashed lines show how the dynamic reply channels from row 2 are passed through the ring connection to row 1, which sends on these channels.

Figure 3. Creation scheme of a torus topology using ring skeletons

of the ring processes when the ring skeleton is called. The first process evaluates the `startRing` function. It creates a dynamic reply channel which is passed through the sequence of ring processes and will be used by the last process to close the ring connection. It is assumed that the number of ring processes is at least two. Thus, the functions `startring` and `unfoldRing` are never called with an empty input list. The input from the parent process is passed through the sequence of ring processes where each ring process takes its part of the input and passes the rest list to its successor process.

The roles of static and dynamic channel connections are exchanged in the two versions of ring skeleton definitions. The previously static output connections to the parent are now modelled by dynamic reply channels while the previously dynamic ring connections can now be realised as static connections, except that the connection from the last to the first ring process is still implemented by a dynamic reply channel.

Experiments with application programs using the ring skeleton show that the recursive ring creation is slightly advantageous as the number of ring processes increases. For a small number of processes there is almost no impact on the runtimes of programs. The number of messages sent and received by the parent process is clearly reduced while the overall amount of messages remains almost the same.

3.2. Torus

The Eden torus skeleton defined in [6] creates all processes by a single process and establishes dynamic interconnection channels between them. In the following, we redefine this skeleton by using the previously defined ring skeletons, as the torus is nothing but a two-dimensional grid with ring connections in both dimensions. Figure 3 depicts the generation scheme for the torus topology used in our redefinition. The first column and all rows are created as unidirectional rings. The other column rings must be installed using dynamic channels.

Figure 4 shows the core of the recursively unfolding torus skeleton. Function `toroideRec` describes the toroid by its dimensions (no. of rows and columns) and the functionality of each node. To place all processes on different processor elements, the first column of the torus structure is created with a variant `ringP` of the recursively unfolding ring skeleton, which allows for placing ring processes with a constant stride. To place processes row by row, the first column is placed with stride `dim2`, i.e. the length of the rows.

```
toroideRec :: (Trans input, Trans output, Trans horiz, Trans vert) =>
  Int -> Int ->                                    -- dimensions
  ((input,horiz,vert) -> (output,horiz,vert)) -> -- node function
  [[input]] -> [[output]]                          -- resulting mapping
toroideRec dim1 dim2 f rows
  = rnf outChans 'seq' start_it 'seq'   -- force channel & ring creation
    list2matrix dim2 outs               -- re-structure output
  where (outChans,outs) = createChans (dim1*dim2)
        ringInput = (list2matrix dim2 outChans, rows)
        -- creating first column ring
        start_it  = ringP dim1 dim2 (\_ -> uncurry zip ) spine
                          (gridlineR dim1 dim2 f) ringInput

-- ring function for 1st column ring
gridRow :: (Trans i, Trans o, Trans h, Trans v) =>
  Int -> Int ->               -- dimensions
  ((i,h,v) -> (o,h,v)) ->     -- node function
  (([ChanName o], [i]), [[ChanName v]]) -> ((), ([[ChanName v]]))
gridRow dim1 dim2 f ((ocs, row), allnextRowChans) =
  let (cChanNamevs, rowChans) = createChans dim2
      -- creating row ring
      start  = startRingDI staticIn (gridNode f) dummyCs mynextRowChans
      staticIn = mzip3 row ocs cChanNamevs
      mynextRowChans = allnextRowChans!!(dim1-2)
      (dummyCs, _ ) = createChans dim2
  in  rnf cChanNamevs 'seq' rnf dummyCs 'seq' start 'seq'
      ((), rowChans:take (dim1-2) allnextRowChans)

-- ring function for row rings
gridNode :: (Trans i, Trans o, Trans h, Trans v) =>
   ((i,h,v) -> (o,h,v)) ->
   ((i,ChanName o, ChanName (ChanName v)),ChanName v,h) -> ((),h)
gridNode f ((a,cResult,cv),cToBottom,fromLeft) =
   new ( \  cFromAbove fromAbove   ->
      let  (out,toRight,toBottom) = f   (a,fromLeft,fromAbove)
      in  parfill cv cFromAbove       -- send vertical input channel
          (parfill cResult out        -- send result for parent
          (parfill cToBottom toBottom -- send data on column ring
          ((), toRight)) ))           -- result and data on row ring
```

Figure 4. Core of Recursively Unfolding Torus Skeleton

The ring function `gridRow` for the first column ring creates a ring for each row. We do not use the normal interface of the `ring` skeleton but the internal startring function, because we want to embed the column processes into the row rings. A subtlety of the inner rings is the circular dependency of their dynamic input, i.e. the dynamic channels to establish the additional column rings. It is necessary to use a variant `startRingDI` which decouples static input, which is available

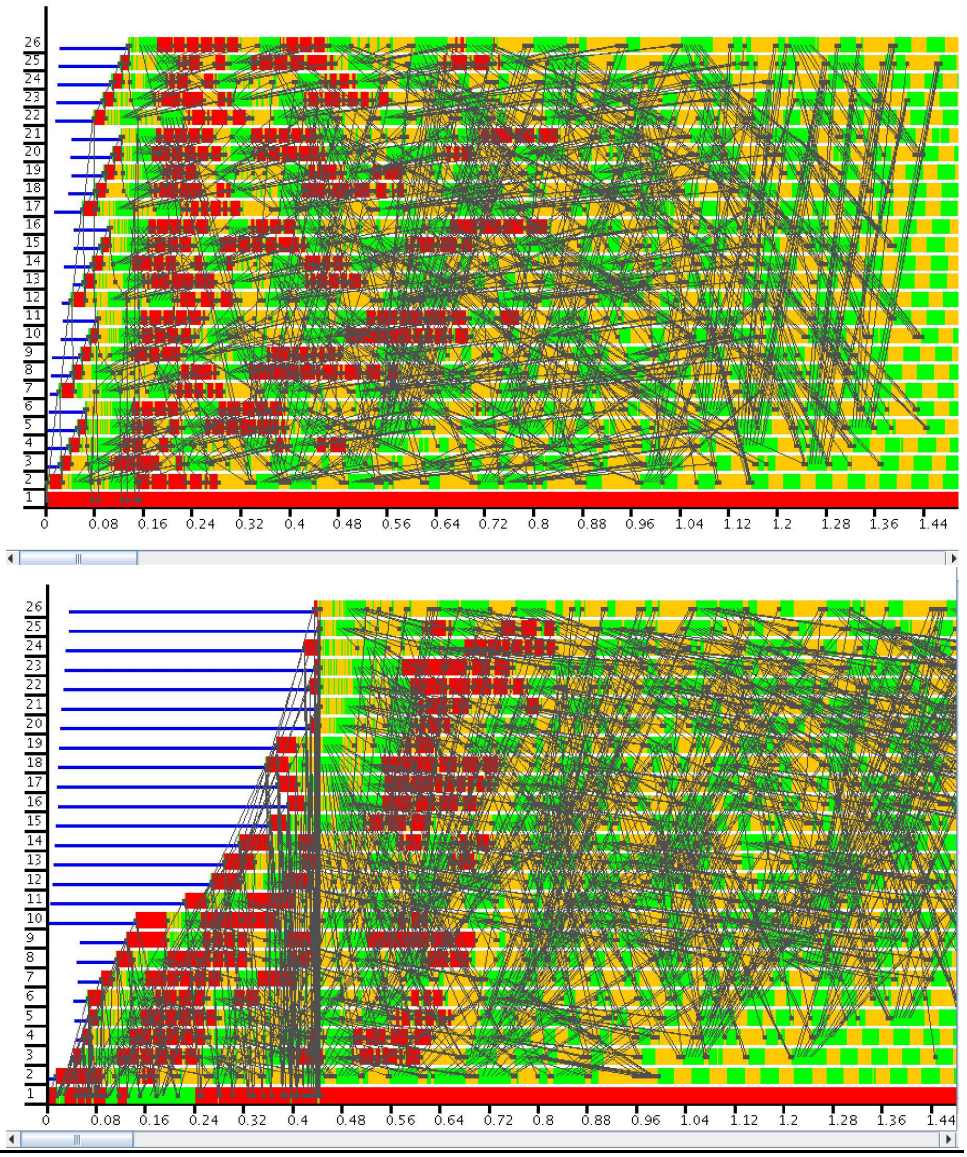


Figure 5. Start Phase of Matrix Multiplication Traces Using Toroid Skeletons, on 26 Processors

at instantiation time, and dynamic input, which will only not be produced after process instantiation. Otherwise, the inner rings would immediately deadlock on process instantiation. Each row ring process returns a channel name for its vertical input, which must be collected and passed to the previous row through the first column ring (as indicated in Figure 3 for the second row).

Measurements with a toroid-based matrix multiplication algorithm (by Gentleman, see [9]) show that runtimes are slightly better for the recursive version, due to a distributed startup sequence. Figure 5 shows traces of the start phase of the matrix multiplication program executed on 26 nodes of a Beowulf cluster, using either the original (single-source) or the recursive skeleton. Processors (resp. processes[1]) are shown as horizontal bars with colour-coded segments for their actions. We distinguish between the states blocked (red – dark grey), runnable (yellow – bright grey) and running (green – middle grey). Idle processors are shown as a smaller bar, and messages between processes are indicated by grey lines.

[1]Because of explicit placement, every processor executes exactly one process. So we identify nodes and processes.

While runtime is only slightly improved, the traces show the expected improvement in startup: process creation is carried out by different processors in a hierarchical fashion in the recursive skeleton implementation. One can observe how the first column unfolds, starting at processor 2 with stride 5, and how each of these processes unrolls one row. Process creation takes about 0.15 sec. in this version, whereas the single-source version below needs 0.4 sec. until all processes start to work (explaining the difference in runtime).

The improvement in startup pays especially for skeletons with a big number of processes. In any case, it substantially reduces the network traffic. The program investigated here already includes the input matrices in the process abstraction instead of communicating these big data structures via channels (which would be more time-consuming). However, the parent process in the single-source version has to send the channel names to each toroid process, which needs 125 messages. The parent process in the recursive version only sends 2 messages – creation and input to the `startring` process of the first column ring.

4. Conclusions and Future Work

The recursive unfolding of rings and toroids spreads the process creation overhead over several processes, thereby avoiding an eventual bottleneck in the originator process. Although the specification and implementation of the recursive unfolding is more sophisticated, case studies have proved that the effort definitely pays off. An interesting aspect which we want to elaborate further is the creation of higher-dimensional topologies by an appropriate nesting of lower-dimensional ones. Currently we have only shown how to define a recursive two-dimensional toroid skeleton using one-dimensional ring skeletons.

References

[1] M. Cole A. Benoit. eSkel – The Edinburgh Skeleton Library, University of Edinburgh 2002. `http://homepages.inf.ed.ac.uk/abenoit1/eSkel/`.

[2] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^3L: A Structured High Level Programming Language and its Structured Support. *Concurrency — Practice and Experience*, 7(3):225–255, May 1995.

[3] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. The MIT Press, Cambridge, MA, 1989.

[4] J. Darlington, Y. Guo, and H.W. To. Structured Parallel Programming: Theory meets Practice. In *Research Directions in Computer Science*. Cambridge University Press, 1996.

[5] H. Kuchen. The Münster Skeleton Library Muesli, University of Münster 2002. `http://www.wi.uni-muenster.de/PI/forschung/Skeletons/`.

[6] R. Loogen, Y. Ortega-Mallén, R. Peña, S. Priebe, and F. Rubio. Parallelism Abstractions in Eden. In *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2003.

[7] R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. Parallel Functional Programming in Eden. *Journal of Functional Programming*, 15(3):431–475, 2005.

[8] R. Plasmeijer, M. van Eekelen, M. Pil, and P. Serrarens. Parallel and Distributed Programming in Concurrent Clean. In *Research Directions in Parallel Functional Programming*, page 323ff. Springer, 1999.

[9] M.J. Quinn. *Parallel Computing*. McGraw-Hill, 1994.

[10] J. H. Reppy. *Concurrent Programming in ML.* Cambridge University Press, August 1999.

[11] P.W. Trinder, K. Hammond, J.S. Mattson Jr., A.S. Partridge, and S.L. Peyton Jones. GUM: a portable implementation of Haskell. In *IFL'95 — Implementation of Functional Languages*, 1995.

Towards Improving Skeletons in Eden*

Mercedes Hidalgo-Herrero[a], Yolanda Ortega-Mallén[b], Fernando Rubio[b]

[a]Dept. Didáctica de las Matemáticas, Universidad Complutense de Madrid, Spain

[b]Dept. Sistemas Informáticos y Programación, Universidad Complutense de Madrid, Spain

The functional language Eden facilitates a skeleton-based methodology for parallel programming. A key point in the efficiency of parallel programs is the distribution of computation among processes. In Eden, this is closely related to its order of evaluation. We describe here an ongoing project whose purpose is to use a prototype implementation of Eden's operational semantics to investigate how alternative evaluation models may either improve or make worse the behavior of the skeletons implemented in Eden.

1. Introduction

The functional parallel language Eden has proven to be highly suitable for a programming methodology based on *algorithmic skeletons*, with the double advantage that skeletons can be implemented and used within the same language. Eden's library provides a rich set of skeletons covering many common parallel patterns such as *parallel map*, *parallel divide-and-conquer*, *parallel search*, and others, as well as typical process topologies like pipelines, grids, rings, and so on [6,14,7,8]. The programmer can either directly use these, or modify them before its use in order to fit better his needs; or even create new skeletons, thus extending the collection.

It is also clear that Eden does not compete for optimal speedups. On the contrary, Eden's strength lies in its higher programming productivity, being its motto: *acceptable speedups at low effort*.

The effectiveness of the use of skeletons depends heavily on the actual implementation of these. Therefore, the majority of skeleton-oriented approaches use low-level languages for the implementation of their skeletons; this should produce accurate and highly efficient implementations, but reduces the flexibility and versatility of the approach, as the set of skeletons usually is fixed. By contrast, Eden offers the possibility of implementing and using skeletons for parallel programming, by considering them as polymorphic higher-order functions. The Eden programmer can choose the process topology and the task granularity, but cannot decide on matters like the placement of processes in processors, or the load balancing strategy. Thus, the efficiency of Eden's skeletons depends on the actual implementation of Eden, that is conditioned by the semantics of the language.

In the present project we desire to investigate alternative semantics for Eden in order to analyze the consequences of some of the decisions adopted during the language design, and in particular how they affect the implementation of skeletons in Eden. For this purpose, it is extremely useful to have a framework where Eden's operational semantics can be easily programmed and that provides mechanisms to reflect changes in the semantics with small effort.

Eden extends the functional language Haskell [13] with coordination features for creating processes with stream-based communication. As a lazy language, Haskell adopts a normal orden of evaluation, avoiding repeated computations by sharing reductions. This lazy approach restricts the exploitation of parallelism because expressions are evaluated only under demand. Therefore, Eden

*Research partially supported by MCyT Spanish project MIDAS: *Metalenguajes para el diseño y análisis integrado de sistemas móviles y distribuidos* (TIC200301000).

overrides the pure lazy approach, combining a non-strict functional application with eager process creation and eager communication. This may produce *speculative* computation, i.e. the calculation of results that may never be used. The amount of speculative computation produced during the evaluation of an Eden program is variable, depending on the number of processors, the speed of basic operations, etc.

The interplay between laziness and eagerness in Eden is precisely established by its operational semantics [3,8]. We are presently developing an interpreter of Eden's operational semantics [4] with the following two main characteristics: (1) different evaluation models for the semantics can be reflected in the implementation with small modifications, and (2) several measures (parallelism, speculative computation, communications) can be taken by modifying some parameters of the semantics.

This interpreter is being implemented in Maude [10,2], a specification language where semantic rules can be represented as rewriting rules; and a *strategy* language [9] is used for controlling the application of the rules.

The rest of the paper is organized as follows. Section 2 gives a brief overview of Eden and its semantics. Section 3 explains the parallel Divide-and-Conquer skeleton and its implementation in Eden. This skeleton is quite simple, but sufficient for showing the kind of analysis that we want to do in the future with more sophisticated skeletons that involve Eden's constructs (streams, dynamic channels, merge process, etc.) that are still to be included in our interpreter. The last section discusses future work.

2. A quick excursion to Eden

We have already mentioned that Eden [1,8] extends the non-strict functional language Haskell with a set of *coordination* features to control parallel evaluation of processes. Coordination in Eden is based on two principal concepts: *explicit definition of processes* and *implicit stream-based communication* [5]. In the same way as there is a distinction between function definition and function application, Eden includes *process abstractions*, i.e. abstract schemes for process behavior, and *process instantiations* for the actual creation of processes.

Moreover, *nondeterminism* is introduced explicitly in Eden by means of a predefined process abstraction which is used to instantiate nondeterministic processes that fairly merge several input streams into a single output stream.

For the purpose of this paper we just concentrate on Eden's essentials, which are captured by the untyped λ-calculus whose abstract syntax is given in Figure 1, where $x \in Var$ represents identifiers and $E \in Exp$ represents expressions.

$E ::= x$		identifier
$\mid$	$\lambda x.E$	λ-abstraction
$\mid$	$E_1 E_2$	application
$\mid$	$E_1 \# E_2$	process instantiation
$\mid$	$\texttt{let}\ \{x_i = E_i\}_{i=1}^{n}\ \texttt{in}\ E$	local declaration

Figure 1. Eden core syntax

When evaluating the expression $E_1 \# E_2$ inside a process p, a new child process q is created which is feeded by its parent process, p, with the value of E_2 via an input channel. Process q evaluates $E_1\ E_2$ and returns the result (to its parent) via an output channel. The diagram in Figure 2 illustrates this behavior.

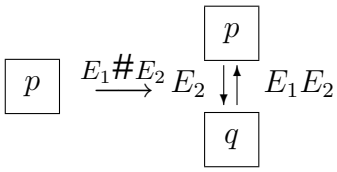


Figure 2. Process creation in Eden

When designing Eden there was great discussion about how to distribute computation between a parent process and its children. In the one extreme the parent would advance as much work as possible, so that every dependent variable of the child process body (or abstraction) should be bound to a *weak head normal form* (whnf) before creating the child process. But this may lead to a poor parallelization, where a process has to do too much computation before delegating work to a subordinate process. In the other extreme —we could say the "laziest"— the parent would pass on all the work to its offspring, so that for an expression $E_1 \# E_2$, the argument E_2 would be evaluated by the parent, while the process abstraction E_1 as well as the application, $E_1\ E_2$, would be evaluated by the new-born child. This may lead to repeated calculations, because certain subexpressions may get evaluated independently by several children of the same parent (as it will illustrated by the example in Section 3). But this can be easily avoided by the programmer, by forcing the evaluation in the parent of these common subexpressions. Therefore, the latter option has been adopted for Eden and its actual implementation, and has been reflected in the operational semantics presented in [8]. Nevertheless, we are interested in analyzing other options; specifically those combinations that are gathered in Figure 3, where EC (*evaluation before copy*) stands for the possibility of evaluating every needed binding before being copied to the initial heap of a newly created process (or the receiver process in the case of a communication); IC (*instantiation copy*) represents the copy of bindings from one process to another corresponding to pending process instantiations; and PAE (*process abstraction evaluation*) indicates the alternatives for the evaluation of a process abstraction in the case of an instantiation: either by the parent process, or by the child.

In the next section we discuss how these combinations affect the behavior of skeletons; in particular, the parallel Divide-and-Conquer scheme.

3. A first case study: Parallel Divide-and-Conquer

As a first case of our analysis we have chosen the parallel Divide-and-Conquer skeleton, a parallelization of the well-known sequential programming scheme. Apart from its simplicity, the main reason for choosing this one is its *task parallel* nature; i.e. it is based on the decomposition of a task into several subtasks to be done in parallel, in contrast to *data parallel* skeletons, where the same operation is applied in parallel to portions of data distributed between processors. The evaluation alternatives for Eden that we have mentioned in the previous section (and that are summarized in

	EC	IC	PAE
1	yes	yes	parent
2	yes	no	parent
3	yes	yes	child
4	yes	no	child
5	no	yes	child
6	no	no	child

Figure 3. Evaluation alternatives

Figure 3) should have a bigger impact on task parallel skeletons, because they affect the work to be done by each process.

Another reason for starting with this skeleton is that it can be easily implemented by creating a dynamic tree of processes where each process is connected to its parent. This hierarchical topology matches perfectly with the process creation and communication mechanisms in Eden, while other skeletons involve special topologies (like pipelines, rings or grids) that are far better implemented in Eden by using dynamic channels[2].

Next we express the Divide-and-Conquer scheme in the restricted syntax given in Figure 1. First we give the sequential version for a split into two subproblems; afterwards we show a straightforward parallel version where every subproblem causes the creation of a process to resolve it.

```
dc = (\trivial.(\solve.(\split.(\combine.(\x.
 (let
   subpr = (split x)
   sol1 = (((((dc trivial) solve) split) combine) (fst subpr))
   sol2 = (((((dc trivial) solve) split) combine) (snd subpr))
  in  (cond (iszero (trivial x))
        (((combine x) sol1) sol2)
        (solve x))
 ))))))


dc_par = (\trivial.(\solve.(\split.(\combine.(\x.
 (cond (iszero (trivial x))
   (let
     subpr = (split x)
     sol1 = (((((dc_par trivial) solve) split) combine) # (fst subpr))
     sol2 = (((((dc_par trivial) solve) split) combine) # (snd subpr))
    in (((combine x) sol1) sol2))
   (solve x))
 )))))
```

In practice, it is more efficient to stop the parallel unfolding before trivial cases are reached; therefore we also present a version where a parameter `depth` determines the maximum level allowed for creating children.

[2]Even though it is possible to create the same topologies without using dynamic channels [12], it is much more complicated and inefficient.

```
dc_par_lim = (\depth.(\trivial.(\solve.(\split.(\combine.(\x.
 (cond (iszero depth)
   (((((dc2 trivial) solve) split) combine) x)
   (cond (iszero (trivial x))
     (let
       subpr = (split x)
       d1 = ((sub depth) one)
       sol1 = (((((dc_par_lim d1 trivial) solve) split) combine) # (fst subpr))
       sol2 = (((((dc_par_lim d1 trivial) solve) split) combine) # (snd subpr))
      in (((combine x) sol1) sol2))
     (solve x)))
 ))))))
```

In the coding we have used functions like `cond` (conditional), `fst` (first) and `snd` (second) for extracting tuple components, `one`, `zero`, `iszero` and `sub` (subtract) that can be easily defined in a λ-calculus (see for instance [11]).

3.1. Discussion on evaluation alternatives

One key design decision for Eden was how to deal with free variables occurring in expressions that should be exported from one process to another —for instance, when creating a new process, or when communicating abstractions—. The alternatives (column EC in Figure 3) are either to evaluate these variables before being copied, or to copy the corresponding evaluation subgraph in the "receiver". The latter may lead to repeat some computations, as it is illustrated with the help of the Divide-and-Conquer skeletons given above. Let us concentrate on the subexpressions

```
(((((dc_par trivial) solve) split) combine) # (fst subpr))
(((((dc_par trivial) solve) split) combine) # (snd subpr))
```

and

```
((((((dc_par_lim depth_1) trivial) solve) split) combine) # (fst subpr))
((((((dc_par_lim depth_1) trivial) solve) split) combine) # (snd subpr))
```

contained in `dc_par` and `dc_par_lim` respectively. The variables `depth_1`, `trivial`, `solve`, `split`, and `combine` are free in each subprocess abstraction. If they are required to be evaluated before the creation of a subprocess, then the evaluation of all these variables is carried out by the parent process and it is done only once. However, if the evaluation is left to the children, the computation is done twice.

On the other hand, the process abstractions

```
((((dc_par trivial) solve) split) combine)
```

and

```
(((((dc_par_lim depth_1) trivial) solve) split) combine)
```

might be bound to a variable in the corresponding let-expression. The relevance of this modification depends on the semantics of the language. The options are gathered in Figure 4; let us analyze them:

1. In this first case the amount of repeated computation is minimized because the abstraction is evaluated once.

2. The evaluation of the abstraction is repeated by each child. In this case it does not matter whether the abstraction is bound to a variable or not.

3. Since the abstraction is not bound to a variable it will be evaluated twice by the parent process.

	Bound to variable	Process abstraction evaluation
1	Yes	Parent
2	Yes	Child
3	No	Parent
4	No	Child

Figure 4. Process abstraction binding and evaluation

4. The evaluation in this last case coincides with that of the second case.

The evaluation of a process abstraction either by the parent or by the children may influence in the speed of evaluation. If the process is created without waiting for the evaluation of the abstraction, then this can be evaluated simultaneously with the expression corresponding to the child input channel, i.e. `(fst subpr)` in our example.

We finish our discussion by remarking that in this example it is irrelevant whether instantiations are copied from one process to another or not (column IC in Figure 3) because there is no interdependence between the abstractions `sol1` and `sol2`.

4. Future work

We have given a first step in our project to analyze the way skeletons are affected by different semantics choices for Eden. However, this analysis has only been made from a theoretical point of view because the actual implementation of Eden's semantics in Maude presents some problems of efficiency that impede us, at the moment, to extract useful measures for our examples. Therefore, our work is still in a preliminary state.

We intend to make a thorough analysis of the different semantic approaches by executing programs based on the skeletons presented in [7]. In order to develop this task, the kernel of Eden presented in this paper needs to be extended. First of all, we have to include streams to represent unbounded communication channels; afterwards, we shall consider dynamic channels, in order to specify more directly non-hierarchical topologies. Finally, we will introduce non-determinism for expressing many-to-one communication, that is essential in many parallel applications, like for instance, in clients-server models.

The analysis will be based on measures of computation such as the number of computation steps, the amount of communication carried out, the number of processes in the final system or the (maximal) amount of thread parallelism that are already included in our Eden interpreter implemented in Maude. In the Divide-and-Conquer skeleton it is easy to work out the number of processes that are created. However, some other measures cannot be obtained unless the computation is effectively made. These measures will help us to analyze the advantages and drawbacks of each semantical option and elucidate whether there is an optimal approach.

References

[1] S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. Eden: Language definition and operational semantics. Technical Report 96/10, Reihe Informatik, FB Mathematik, Philipps-Universität Marburg, Germany, URL `http://www.mathematik.uni-marburg.de/~eden/`, 1996.

[2] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude Manual (Version 2.1)*, March 2004. URL `http://maude.cs.uiuc.edu/manual`.

[3] M. Hidalgo-Herrero and Y. Ortega-Mallén. An operational semantics for the parallel language Eden. *Parallel Processing Letters (World Scientific Publishing Company)*, 12(2):211–228, 2002.

[4] M. Hidalgo-Herrero, Alberto Verdejo, and Y. Ortega-Mallén. Looking for eden through maude and its strategies. In *Proc. of the 5th Jornadas sobre Programacin y Lenguajes*, 2005. To appear.

[5] G. Kahn and D. MacQueen. Coroutines and networks of parallel processes. In *IFIP'77*, pages 993–998. Eds. B. Gilchrist. North-Holland, 1977.

[6] Ulrike Klusik, Rita Loogen, Steffen Priebe, and Fernando Rubio. Implementation skeletons in eden: Low-effort parallel programming. In *Proceedings of the 12th International Workshop on Implementation of Functional Languages, (IFL'00 selected papers)*, pages 71–88. LNCS 2011, Springer, 2001.

[7] R. Loogen, Y. Ortega-Mallén, R. Pea, S. Priebe, and F. Rubio. *Patterns and Skeletons for Parallel and Distributed Computing*, chapter 4: Parallelism Abstractions in Eden, pages 95–128. Eds. F. A. Rabhi and S. Gorlatch. Springer, 2002.

[8] R. Loogen, Y. Ortega-Mallén, and R. Peña. Parallel functional programming in Eden. *Journal of Functional Programming*, 15(3):431–475, 2005.

[9] N. Martí-Oliet, J. Meseguer, and A. Verdejo. Towards a strategy language for Maude. In N. Martí-Oliet, editor, *Proceedings Fifth International Workshop on Rewriting Logic and its Applications, WRLA 2004, Barcelona, Spain, March 27 – April 4, 2004*, volume 117 of *Electronic Notes in Theoretical Computer Science*, pages 417–441. Elsevier, 2004.

[10] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[11] G. Michaelson. *An Introduction to Functional Programming through Lambda Calculus*. Addison-Wesley, 89.

[12] R. Peña, F. Rubio, and C. Segura. Deriving non-hierarchical process topologies. In *Trends in Functional Programming (Selected papers of the 3rd Scottish Functional Programming Workshop)*, volume 3, pages 51–62. Intellect, 2002.

[13] S. Peyton Jones. *Haskell 98 language and libraries: the Revised Report*. Cambridge University Press, 2003.

[14] F. Rubio. *Programación funcional paralela eficiente en Eden*. PhD thesis, Dept. Sistemas Informáticos y Programación, Universidad Complutense de Madrid, 2001.

850

Reasoning About Skeletons in Eden

Ricardo Peña[a], Clara María Segura[a]

[a]Departamento Sistemas Informáticos y Programación, Universidad Complutense de Madrid, Spain

1. Introduction

The parallel-functional language Eden extends the lazy functional language Haskell by constructs to explicitly define and launch processes. Skeletons can be implemented in the language itself and can be invoked in parallel applications as higher-order functions. So, Eden can be used both as a *system-level* implementation language and as an *application-level* parallel one. In order to be useful, skeletons should be proved correct and should lead to an efficient use of the underlying machine.

The paper presents examples of system-level programming in Eden, showing the conciseness of defining skeletons at a high level of abstraction. Efficiency reasoning is provided by accurate cost models and correctness is proved by induction proofs. If the parallel program uses finite data structures, the proof implies also successful *termination*. If it uses infinite ones (i.e. streams), the proof implies *productivity*. A program is productive if, whenever it receives continuous input, it provides continuous output. Productivity amounts to deadlock freedom. The paper summarizes the correctness and efficiency results contained in some other papers, specially [5–8] and [4].

The plan is as follows: in Section 2, we briefly describe Eden's syntax and semantics; Section 3 shows examples of system level programming by defining two parallel implementations of the skeleton `map`, providing their respective cost models and proving these implementations correct; Section 4 survey other Eden skeletons in less detail. Finally, Section 5 provides some conclusions and related work.

2. Eden features summary

A *process abstraction* is just the application of the predefined function `process` to a function. If f id of type `a -> b` then `process f` is of type `Process a b`. It defines the behaviour of a process receiving `x::a` as input and returning `f x::b` as output. A *process instantiation* uses the predefined infix operator:

```
(#) :: (Transmissible a, Transmissible  b) => Process a b -> a -> b
```

In order to be able to transmit a value, its type must belong to the type class `Transmissible`. The evaluation of an expression `(process f) # e2` leads to the dynamic creation of a process together with its interconnecting communication channels. The instantiating or *parent process* will be responsible for evaluating and sending `e2` via an implicitly generated channel, while the new *child process* will evaluate the application `f e2` and return the result via another implicitly generated channel. For input or output tuples, independent concurrent threads and channels are created to evaluate each tuple component.

Once a process is running, only fully evaluated data objects are communicated. The only exceptions are lists, which are transmitted in a *stream*-like fashion, i.e. element by element. Each list element is first evaluated to full normal form and then transmitted. Concurrent threads trying to access input not yet available, are temporarily suspended. This is the only way in which Eden processes synchronize. Let us remark that there are no explicit instructions handling channels or messages as communication/synchronization is completely implicit.

Replacing in the definition of `map`

```
map       :: (a -> b) -> [a] -> [b]
map f xs  =  [f x | x <- xs]
```

the function application `f x` by a process instantiation, leads to a simple parallel `map` skeleton in which a different process is launched for each element of the input list:

```
map_par :: (Transmissible a, Transmissible b) => (a -> b) -> [a] -> [b]
map_par f xs =  [process f # x | x <- xs] `using` spine
```

The `spine` strategy (see [11]) is used to eagerly evaluate the spine of the process instantiation list. Otherwise, the laziness of Haskell would prevent that all processes were immediately created.

Many-to-one communication is an essential feature for some parallel applications, but it spoils the purity of functional languages, as it introduces non-determinism. In Eden, the predefined process abstraction

```
merge :: Transmissible a => Process [[a]] [a]
```

is used to instantiate a process which fairly merges a list of input streams into a single (non-deterministic) output stream. The incoming values are passed to the output stream in the order in which they arrive. In this way `merge` provides many-to-one reactive communication. It can profitably be used to quickly react to requests coming in an unpredictable order from a set of processes. Even though the skeletons presented are deterministic, some of them are required to immediately react to requests for work coming from a group of worker processes. An instantiation of `merge` will propagate these requests as they are being produced.

Eden's compiler[1] has been developed by extending the Glasgow Haskell Compiler (GHC) [9]. Eden's runtime system (RTS) is an implementation of the DREAM abstract machine [1] on top of a message passing library. Both PVM and MPI can be used. Therefore, the compiler can be ported to any architecture where GHC and either PVM or MPI are available.

Eden provides no *placement annotations*. However, Eden's RTS supports two modes to map processes to processors, which can be chosen by the user for each execution. *Round-robin mode*: If several processes are instantiated from a particular processor p, they are mapped to consecutive processors starting with the one numbered one more than p. *Random mode*: Each processor maps instantiated processes to randomly chosen processors.

The number of processors is provided by the integer constant `noPe`. It can be used to adapt the number of processes to the number of available processors.

3. The `map` skeleton in Eden

The definition of a skeleton consists of two parts: (1) A *specification* giving the type of the skeleton and a description of its observable behavior as a higher-order function; and (2) the *implementations*. A skeleton may have several implementations, and for each one two pieces must be provided: A parallel program (the *algorithm*), and a formula describing its expected parallel execution time (the *cost model*).

Cost models describe the parallel time of the algorithm. For a survey of cost models see e.g. [2]. The cost models presented in this section are an adaptation to Eden of classical cost models

[1] Available at http://www.mathematik.uni-marburg.de/inf/eden

Problem dependent parameters	
N	Size of the input
t_f	sequential CPU time for function f
nwI	number of words of input message going to a child
nwO	number of words of output message coming from a child
RTS dependent parameters	
t_{create}	CPU time in a parent processor to create a child process
$t_{\#}$	CPU time in a child processor to create a new process
Architecture dependent parameters	
P	Number of processors
δ	latency of a message, from start sending to start receiving
λ	start-up fixed CPU cost for sending or receiving a message
β	per-word CPU cost for sending or receiving a message

Figure 1. Parameters of the cost models

appearing in the literature. In Figure 1 we show the different parameters involved in Eden skeletons cost models. We will use the abbreviations:

$$t_{unpackI} = t_{packI} = \lambda + \beta nwI$$
$$t_{unpackO} = t_{packO} = \lambda + \beta nwO$$

In the rest of this section we concentrate on two different implementations of the skeleton map. For each one we present its cost model and a proof of its correctness.

For proving correctness, we will use induction on natural numbers and $P(n)$ will denote a predicate where the natural n will be related to the length of one or more lists in the program being verified. Our aim is to prove $\forall n.P(n)$ using the ordinary induction rule:

$$\frac{P(0) \quad \forall n \geq 0.P(n) \Rightarrow P(n+1)}{\forall n.P(n)}$$

When the rule is applied to a function on finite lists, it proves the total correctness of the function, i.e. the proof implies termination. When the rule is applied to a recursively defined list, it proves that its length increases at each recursive call, i.e. the list is potentially infinite, and then the function producing it is productive.

In what follows, $|xs|$ denotes the length of the list *xs* and xs_i the ith element of the list *xs* starting from 0. For instance, the specification of `map` can be done by the following predicate:

$$P_{map}(n) \stackrel{\text{def}}{=} \forall f.\forall xs.\ |xs| = n \Rightarrow \texttt{map f } xs = [f\ x_i \mid x_i \leftarrow xs]$$

It is trivial to prove its correctness by using the recursive definition of `map`. Notice that the proof implies termination of `map` for finite lists xs and implies productivity for infinite ones.

3.1. Farm Implementation

In the farm implementation of `map`, called `map_farm`, a single process is created in each available processor and tasks are evenly distributed into processors. Funtion f is applied to each task and results are collected by the parent process. If the parent process load is low, we locate it in the same processor as one of its children. Otherwise, we devote a separate process for the parent. A threshold parameter is used to allow the skeleton to take this decision. This implementation is appropriate when task granularity is uniform and an even distribution of the number of tasks amongst all the processors is desired. The length of the task list must be rather higher than the number of available

processors in order to improve the load balance. In order to instantiate processes correctly, the round-robin mode is used by the RTS. The distribution and collection functions are also parameters of the skeleton. Here is the implementation:

```
map_farm :: (Transmissible a,Transmissible b) => Int -> (a -> b) -> [a] -> [b]
map_farm thr = farm np unshuffle shuffle  where np | noPe > thr = noPe - 1
                                                   | otherwise  = noPe
farm :: (Transmissible a, Transmissible b) =>
        Int -> (Int -> [a] -> [[a]]) ->([[b]] -> [b]) -> (a -> b) -> [a] -> [b]
farm np unshuffle shuffle f tasks = shuffle (map_par (map f) (unshuffle np tasks))
```

Different strategies to split the work into the different processes can be used provided that, for every list `xs`, `(shuffle . unshuffle n) xs == xs` holds. As this is standard Haskell programming, we do not show these functions here. The cost model for `map_farm` is the following:

$$\begin{array}{lcl} t_{map\_farm} & = & L_{init} + t_{worker} + L_{final} \\ L_{init} & = & P(t_{create} + t_{packI} + t_{unshuffle_1}) + \delta \\ L_{final} & = & \delta + t_{unpackO} + t_{shuffle_1} \\ t_{worker} & = & t_{\#} + \lceil \frac{N}{P} \rceil (t_{unpackI} + t_f + t_{packO}) \end{array}$$

In essence, $t_{map\_farm} = k_1 P + k_2 \lceil \frac{N}{P} \rceil + k_3$ for some constants k_1, k_2 and k_3. In [5], some actual executions are shown accurately agreing with this model.

In order to proof the correctnes of this implementation, we need to prove:

$$P_{map\_farm}(n) \stackrel{\text{def}}{=} \forall f. \forall \mathtt{xs}.\ |\mathtt{xs}| = n \Rightarrow \mathtt{map\_farm\ f\ xs} = [f\ x_i \mid x_i \leftarrow \mathtt{xs}]$$

We assume that the following predicate has been proved by induction on n:

$$P_{unshuffle}(n, np) \stackrel{\text{def}}{=} \forall n. \forall np. \forall xs.\ |\mathtt{xs}| = n \Rightarrow \mathtt{unshuffle\ np\ xs} = \mathtt{xss} \mid concat\ xss \in perm(xs)$$

where $perm$ gives the set of permutations of a list. We also assume that `map_par` satisfies the predicate $P_{\mathtt{map}}(n)$ of `map`. Then, we have the following equivalences:

$$\begin{array}{ll} & \mathtt{map\_par}\ (\mathtt{map}\ f)\ (\mathtt{unshuffle\ np\ xs}) \\ = & \quad \{\text{by the correctness of } \mathtt{map\_par}\} \\ & [\mathtt{map}\ f\ \mathtt{ys} \mid \mathtt{ys} \leftarrow \mathtt{unshuffle\ np\ xs}] \\ = & \quad \{\text{by the correctness of } \mathtt{map}\} \\ & [[f\ y \mid y \leftarrow \mathtt{ys}] \mid \mathtt{ys} \leftarrow \mathtt{unshuffle\ np\ xs}] \\ = & \quad \{\text{by the correctness of } \mathtt{map}\} \\ & [[z \mid z \leftarrow \mathtt{map\ f\ ys}] \mid \mathtt{ys} \leftarrow \mathtt{unshuffle\ np\ xs}] \\ = & \quad \{\text{by the predicate } P_{unshuffle}(n, np)\} \\ & [[z \mid z \leftarrow \mathtt{zs}] \mid \mathtt{zs} \leftarrow \mathtt{unshuffle\ np\ (map\ f\ xs)}] \\ = & \quad \{\text{by change of notation}\} \\ & \mathtt{unshuffle\ np\ (map\ f\ xs)} \end{array}$$

Then, by using the property `(shuffle . unshuffle np) xs == xs` we finally have:

$$\begin{array}{ll} & \mathtt{map\_farm\ f\ xs} \\ = & \mathtt{farm\ np\ unshuffle\ shuffle\ f\ xs} \\ = & \mathtt{shuffle\ (map\_par\ (map\ f)\ (unshuffle\ np\ xs))} \\ = & \mathtt{shuffle\ (unshuffle\ np\ (map\ f\ xs))} \\ = & \mathtt{(shuffle\ .\ unshuffle\ np)\ (map\ f\ xs)} \\ = & \mathtt{map\ f\ xs} \end{array}$$

3.2. Replicated Workers Implementation

The load balance obtained using the farm scheme can be poor when the granularity of the tasks is not uniform. Instead of using a fixed task distribution scheme, we can distribute work on demand.

This gives rise to the *replicated workers* implementation of `map`. Initially, the manager assigns two tasks to each of the workers. By assigning more than one task, the idle time between tasks is minimized. Each time a worker finishes a task, it sends an acknowledgment message to the manager including the task result, and then a new task is assigned to that process. The computation finishes when the manager has received all the task results. By using the process `merge`, acknowledgments from different processes can be received by the manager as soon as they are produced. If each acknowledgment contains the identity of the sender process, the list of merged results can be inspected in order to know who has sent the message, and then a new work can be assigned to it. In Eden, this solution can be expressed as a set of mutually recursive list definitions:

```
map_rw f ts = let tids = zip [0..] ts in
 letrec  outs =  let urs = merge # outs
                     reqs= [0..np-1] ++ [0..np-1] ++ map first urs
                     ins = distribute tids reqs
                 in  [(worker f i) # in | (i,in) <- zip [0..] ins]
 in sortMerge outs

worker f i =  process (\ts -> map (\(it,t) -> (i,it,f t)) ts)
```

The list `tids` just assign a different number to each task. The list of lists `outs` contains a result list coming from each worker. The list `urs` produced by `merge` can be understood as the temporal sequence of requests for new tasks. Function `distribute` assigns a new task form the list `tids` to the worker who has first sent a request. The implementation of `distribute` and `sortMerge` are not shown. They are assumed to satisfy the following predicates:

$$P_{distribute} \stackrel{\text{def}}{=} |xs| \leq |rs| \wedge \forall i \in \{0..\,|rs|-1\}.rs_i \in \{0..np-1\} \Rightarrow$$
$$\texttt{distribute xs rs} = \texttt{yss} \mid \; |yss| = np \wedge (\forall j \in \{0..np-1\}.ordered(yss_j))$$
$$\wedge \textstyle\sum_{i=0}^{np-1} |ys_i| = |xs| \wedge concat\ ys \in perm(xs)$$
$$P_{sortMerge} \stackrel{\text{def}}{=} \texttt{sortMerge xss} = \texttt{rs} \mid rs = map\ third\ xs \wedge ordered(xs) \wedge xs \in perm(concat\ xss)$$

That is, `distribute` behaves rather similarly to `unshuffle` and `sortMerge` produces an ordered list from a list of ordered lists. If the number N of tasks is much greater than the number P of processors, the cost model for `map_rw` is:

$$\begin{aligned} t_{map\_rw} &= L_{init} + t_{worker} + L_{final} \\ L_{init} &= P(t_{create} + t_{packI} + t_{distribute_1}) + \delta \\ L_{final} &= \delta + t_{unpackO} + t_{sortMerge_1} \\ t_{worker} &= t_{\#} + \tfrac{N}{P}(t_{unpackI} + t_{comp} + t_{packO}) \end{aligned}$$

Assuming a perfect load balance, it can be considered that every worker receives the exact average number of tasks, each task costing the average computing cost $t_{comp} = \frac{1}{N}\sum_{i=1}^{N} t_{f_i}$, where t_{f_i} represents the cost of function f when applied to task i. In $t_{distribute_1}$ we consider accumulated the costs of `zip, ++` and `map first` functions when producing one element. Notice that the ceiling operation has disappeared from $\frac{N}{P}$. Then, the simplified model is $t_{map\_rw} = k_1 P + k_2 \frac{N}{P} + k_3$. This is very similar to that of t_{map_farm} but now tasks of different granularities are allowed. Again, [5] shows examples accurately agreeing with this model.

In order to prove the correctness of the skeleton the following predicates are proposed for each auxiliary list produced by the algorithm:

$$P_{outs}(ts, f, n) \stackrel{\text{def}}{=} \exists n_0 \ldots n_{np-1}.n = \sum_{i=0}^{np-1} n_i \wedge \forall i \in \{0..np-1\}.n_i = |outs_i| \wedge ordered(outs_i)$$
$$\wedge (\forall j \in \{0..n_i - 1\}.r = f\ ts_{it} \wedge iw \in \{0..np-1\} \quad where\ \ (iw, it, r) = (outs_i)_j\ )$$
$$P_{urs}(n) \stackrel{\text{def}}{=} \forall i \in \{0..n-1\}.iw \in \{0..np-1\} \quad where\ \ (iw, \_, \_) = urs_i$$
$$P_{reqs}(n) \stackrel{\text{def}}{=} \forall i \in \{0..n-1\}.reqs_i \in \{0..np-1\}$$
$$P_{ins}(ts, n) \stackrel{\text{def}}{=} \exists n_0 \ldots n_{np-1}.n = \sum_{i=0}^{np-1} n_i \wedge \forall i \in \{0..np-1\}.n_i = |ins_i| \wedge ordered(ins_i)$$
$$\wedge (\forall j \in \{0..n_i - 1\}.reqs_{it} = i \wedge t = ts_{it} \quad where\ (it, t) = (ins_i)_j\ )$$

The first one expresses that `outs` consists of np lists whose lengths sum is the number n of tasks. Each sublist consists of triples (iw, it, r) and is ordered by task identity it. The worker identities iw are numbers in the range $0..np-1$, and values r are the result of applying f to the task in position it of list ts. The only important property of the second predicate is that the first component of each triple of `urs` is an actual worker identity in the range $0..np-1$. A similar comment applies to `reqs`. The last predicate expresses that `ins` consists of np lists whose lengths sum is n. Each sublist consists of tuples (it, t) and is ordered by task identity it. It also says that tasks are located in `ins` according to the requests in list `reqs`. The main theorem to be proved is that `map_rw` behaves as `map`:

$$P_{map\_rw}(n) \stackrel{\text{def}}{=} \forall f.\forall xs.\ |xs| = n \Rightarrow \texttt{map\_rw f}\ xs = [f\ x_i \mid x_i \leftarrow xs]$$

We first prove the following predicate $Q(n)$ by induction on the length n of the task list:

$$Q(n) \stackrel{\text{def}}{=} \forall f.\forall ts.\ |ts| = n \Rightarrow P_{outs}(ts, f, n)$$

The proof scheme is as follows:

Base case $n = 0 \Rightarrow ts = [\,] \Rightarrow P_{outs}([\,], f, 0)$

Inductive step Assuming a task list ts of length $n+1$ and the induction hypothesis $P_{outs}(ts', f, n)$ for a list ts', the prefix of lenght n of ts, we easily prove from the program:

$$P_{outs}(ts', f, n) \Rightarrow P_{urs}(n) \Rightarrow P_{reqs}(n + 2np)$$

Assuming $P_{reqs}(n + 2np)$, $P_{distribute}(tids, reqs)$ and $|\,tids\,| = n + 1$, by the definition in the program of `ins`, `outs` and `worker`, we have:

$$P_{reqs}(n + 2np) \wedge |tids| = n + 1 \Rightarrow P_{ins}(ts, n + 1) \Rightarrow P_{outs}(ts, f, n + 1)$$

and then we are done.

This amounts to proving $\forall n.Q(n)$. Knowing that `map_rw f ts = sortMerge outs` and assuming $P_{sortMerge}(outs)$, it is straightforward to prove $\forall n.P_{map\_rw}(n)$, i.e. `map_rw` terminates and is correct for all finite lists.

Productivity is an added value of some programs. It says that, even when the program does not terminate, it will produce useful output during its work by processing increasing portions of its infinite input. For instance, a function reversing a list is terminating for finite input but non

productive for an infinite one. Our `map_rw` skeleton produces an infinite result for an infinite number of tasks. For lack of space, we only point out some key ideas about the proof:

First, the predicates $P_{outs}(ts, f, n)$, $P_{distribute}$ and $P_{map\_rw}(n)$ must be slightly modified. The new definitions will allow that list `outs`, the output sublists produced by `distribute` and the list produced by `map_rw` may have any number of elements. Predicates $P_{distribute}$ and P_{map_rw} will have an additional parameter n indicating the number of tasks processed up to some point in time. Predicates $oredered$, $perm$ and $concat$ will be modified accordingly. The main theorem is now:

$$|ts| = \omega \Rightarrow \forall n. \forall f. P_{outs}(ts, f, n)$$

Again the proof is done by induction on n. The key step is to show that $P_{outs}(ts, f, n + 2np) \Rightarrow P_{outs}(ts, f, n + 1)$, which can be done by reasoning that $[ts_0..ts_n]$ is a prefix of $[ts_0..ts_{n+2np-1}]$. This will show that the list `outs` is productive. By separately proving that `sortMerge` is productive, also will do `map_rw`. More details can be found in [6] and [8].

4. Other Eden Skeletons

More examples of Eden skeletons can be found in [5,4]. For instance, there is a *divide and conquer* skeleton where a dynamic tree of processes is created in which each process is connected to its parent. An integer parameter determines the maximum level after which no more children processes are generated, and the sequential version is used instead. The implementation is as follows:

```
dc_naive ::(Transmissible a, Transmissible b) =>
    Int -> (a -> Bool) -> (a -> b) ->( a -> [a]) -> (a -> [b] -> b) -> a -> b
dc_naive 0 trivial solve split combine  = dc trivial solve split combine
dc_naive d trivial solve split combine x
  | trivial x   = solve x
  | otherwise   = combine x c
 where c = map_par (dc_naive (d-1) trivial solve split combine) (split x)
```

Notice here that there is no single manager process, as it was the case in the `map` skeletons, because every child is a parent process of the next process level. A better implementation is obtained by first flattening the task tree and producing a list of tasks. Then the `map_rw` skeleton is used to solve the tasks and finally a single solution is computed from the list of results.

A pipeline skeleton consists of a list of stages. Each stage applies a different function to the result obtained in the previous one. A naïve parallelization of this scheme can be done by instantiating a different process to evaluate each of the pipeline stages. This can be expressed in Eden as follows:

```
pipe_naive :: Transmissible a => [[a]->[a]] -> [a] -> [a]
pipe_naive fs xs  = (ppipe fs) # xs

ppipe :: Transmissible a => [[a]->[a]] -> Process [a] [a]
ppipe [f]    = process f
ppipe (f:fs) = process (\xs -> (ppipe fs) # (f xs))
```

This definition does not achieve the desired topology because the last process of the pipe cannot sends the values directly to the main process. Instead, a hierarchical topology is created. In order to get a flat one, a better definition makes use of Eden's *dynamic channels* facility, not explained in this paper (see for instance [4]). In [7] a manual transformation was proposed which allows to obtain a flat solution based on dynamic channels by using as specification a hierachical one. The method has been also applied to ring, grid and torus skeletons. Cost models and proof of correctness have been produced for most of these implementations (confirm in [5,8]).

5. Conclusions

The main differences between Eden and more traditional skeleton-based languages are two: (1) Eden is functional while the vast majority of skeleton implementation languages are imperative, and (2) skeletons can be implemented and used within the same language. In other approaches, skeletons are often implemented in a low-level language different from the one in which they are used. For instance, in *PMLS* [10] Scaife et al. extend an ML compiler by machinery which automatically searches the given program for higher-order functions which are suitable for parallelisation. During compilation these are replaced by efficient low-level implementations written in C and MPI.

The main point of this paper has been showing that the conciseness of the functional notation allows to conduct formal proofs of the skeletons with a reasonable effort. Both equational and predicate based reasoning have been used, and both termination and productivity of skeletons have been shown by induction on natural numbers. In [8] Eden skeletons were proved terminating and/or productive by using an extension of the sized type theory by Hughes and Pareto [3].

Additionaly, cost models have been defined from the functional implementations in which low level runtime system an arquitecture parameters appear. These cost models allow the programmer to reason about the runtime behaviour of the corresponding parallel programs.

References

[1] S. Breitinger, U. Klusik, R. Loogen, Y. Ortega-Mallén, and R. Peña. DREAM: the Distributed Eden Abstract Machine. In *IFL'97, Selected Papers. LNCS 1467*, pages 250–269. Springer-Verlag, 1998.

[2] M. Hamdan. *A Combinational Framework for Parallel Programming Using Algorithmic Skeletons*. PhD thesis, Department of Computing and Electrical Engineering. Heriot-Watt University, 2000.

[3] J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *Conference Record of POPL '96: The* 23^{rd} *ACM SIGPLAN-SIGACT*, pages 410–423, 1996.

[4] R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. Parallel-Functional Programming in Eden. *Journal of Functional Programming*, 15(3):431–475, 2005.

[5] R. Peña and F. Rubio. Parallel Functional Programming at Two Levels of Abstraction. In *Principles and Practice of Declarative Programming PPDP'01. Florencia (Italy). ACM Press*, pages 187–198, September 2001.

[6] R. Peña, F. Rubio, and C. Segura. Convenience, Efficiency and Correctness in the Parallel Functional Language Eden. In *Integrated Design and Process Technology, IDPT'02. Pasadena (EE.UU.)*, pages 1–10, June 2002.

[7] R. Peña, F. Rubio, and C. Segura. Deriving Non-Hierarchical Process Topologies. In *Trends in Functional Programming 3. Selected Papers of the 3rd Scottish Functional Programming Workshop, SFP'01. Intellect*, pages 51–62, 2002.

[8] R. Peña and C. Segura. Sized Types for Typing Eden Skeletons. In *Selected papers of Implementation of Functional Languages, IFL 2001*, pages 1–17. LNCS 2312. Springer, 2002.

[9] S. L. Peyton Jones. Compiling Haskell by Program Transformations: A Report from the Trenches. In *ESOP'96, LNCS 1058*, 1996.

[10] N. Scaife, G. Michaelson, and S. Horiguchi. Comparative Cross-Platform Results from a Parallelizing SML Compiler. In *EuroPar 2001*, Manchester, England, 2001.

[11] P. W. Trinder, K. Hammond, H. W. Loidl, and S. L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1), 1998.

Merging Compositions of Array Skeletons in SAC

Clemens Grelck[a], Sven-Bodo Scholz[b]

[a]University of Lübeck, Institute of Software Technology and Programming Languages, Ratzeburger Allee 160, 23538 Lübeck, Germany

[b]University of Hertfordshire, Department of Computer Science, College Lane, Hatfield, Hertfordshire, AL10 9AB, United Kingdom

In the array language SAC, implicitly parallel array skeletons can be defined from meta-skeletons that are part of the language itself. This offers a high potential for code reuse as well as for high-level program specifications. However, the downside of introducing parallelism on the level of relatively simple application building blocks is limited runtime performance due to a generally poor ratio between productive computation on the one side and communication and synchronization on the other side. To overcome this limitation we have developed compiler optimizations that identify different composition scenarios of skeletons each of which is addressed by a specific code transformation technique that merges individual skeletons into a single more powerful one. This paper demonstrates how these transformations can be combined to systematically restructure compositions of skeletons into more complex skeletons that, in turn, can be compiled into efficiently executable parallel code.

1. Introduction

SAC (Single Assignment C) [12] is an implicitly parallel, purely functional array language designed with numerical applications in mind. The language design of SAC aims at combining generic, high-level specifications of array-based algorithms with a runtime performance that is competitive with low-level, machine-oriented languages. Compiler-directed parallelization of SAC programs [9] allows shared memory multiprocessor machines to be utilized without any additional programming effort.

Parallelism in SAC programs stems from the use of array skeletons like maps of scalar operations, rotation and shifting operations, subarray selections, or reduction operations. In SAC, all these skeletons are defined in the language itself by means of meta-skeletons, named WITH-loops. As a consequence, SAC programs usually consist of various layers of skeleton composition. This programming style leads to highly generic code with good opportunities for reuse on each layer.

The downside of this approach is that it also leads to deeply nested compositions of WITH-loops that are computationally light-weight. With parallel program execution bound to individual WITH-loops, the ratio between productive computation and synchronization/communication overhead tends to be unfavorable. In order to overcome this limitation we have developed optimization techniques that systematically restructure compositions of WITH-loops from a representation that is amenable to code development and maintenance towards a representation that is suitable for efficient parallel execution.

We have identified three different types of composition: vertical, horizontal, and nested composition. Vertical composition describes computational pipelines where the result of one WITH-loop becomes the argument of another WITH-loop. In order to avoid this kind of composition, the computational pipelines need to be shifted from the level of entire arrays to the level of individual elements. This is achieved by an optimization technique called WITH-loop-folding [11]. In essence, it performs forward substitutions from one WITH-loop body of a computational pipeline into the body

of the subsequent WITH-loop until the first WITH-loop becomes obsolete.

Horizontal composition is characterized by multiple WITH-loops that compute separate results from the same or at least an overlapping set of arguments. Provided there are no data dependencies between the WITH-loops under consideration, these can be combined into a single WITH-loop that computes all results simultaneously. As this technique to some extent resembles classical loop fusion techniques it is called WITH-loop-fusion. When applied, it results in multi-operator WITH-loops. They represent multiple skeleton operations and compute several arrays in a single step.

Nested composition occurs whenever individual elements of a WITH-loop-defined array are defined by yet another WITH-loop. Under certain circumstances such nestings can be combined into single WITH-loops that operate one scalar values eliminating all micro-synchronizations that would result from a naive compilation otherwise. A transformation scheme to this effect is WITH-loop-scalarization [10].

While folding, fusion, and scalarization are orthogonal in the intermediate code scenarios they address, all three improve the quality of parallelized code by reducing the need for synchronization and communication. By systematically merging computationally light-weight array skeletons into more complex operations they also improve the quality and the efficiency of scheduling workload to processing units. In this paper we demonstrate their combined effect on the optimization and parallelization of intermediate SAC code.

The remainder of this paper is organized as follows. Section 2 gives a rough idea of the design of SAC and provides a brief introduction into WITH-loops. In Section 3, we introduce a running example. Sections 4, 5, and 6 sketch the three optimization techniques and discuss their effect on the running example. Related work is presented in Section 7 and Section 8 concludes.

2. With-loops in SAC

As the name suggests, SAC can be considered a functional variant of C. The basic idea is to restrict C in a way that guarantees a side-effect free setting. Essentially, this is achieved by eliminating global variables and pointers from C and by having a clear separation between expressions and assignments. All major language constructs from C such as conditionals, loops, assignments, function definitions, or function calls can be adopted without any change in their operational behavior (for details see [12]).

On top of this language core SAC supports arrays as the only data structure. For manipulating these, several variants of meta-skeletons, the so called WITH-loops, are provided. As they are first class citizens of the language, i.e., they can be used in any expression position, all basic skeletons for array manipulation such as maps of scalar operations, rotation and shifting operations, can be defined in terms of WITH-loops.

For the purpose of this paper we consider WITH-loops program expressions that take the general form

with (*lower* <= *idx_vec* < *upper*) : *expr* ;
genarray(*shape*, *default*)

where *idx_vec* is an identifier, *lower*, *upper*, and *shape* denote expressions that should evaluate to vectors of identical length, and *expr* and *default* denote arbitrary expressions. Such a WITH-loop defines an array of shape *shape*, whose elements are either computed by the expression *expr* or by the default expression *default*. Which of these two values is chosen for an individual element depends on its location, i.e., it depends on its index position. If the index is within the range specified by

the lower bound *lower* and the upper bound *upper*, *expr* is chosen, otherwise *default* is taken. As a simple example, consider the WITH-loop

```
with ([1] <= iv < [4]) : 2;
genarray( [5], 0)
```

It computes the vector `[0, 2, 2, 2, 0]`. Note here, that the use of vectors for the shape of the result and the bounds of the index space (also referred to as the **generator**) allows WITH-loops to denote arrays of arbitrary rank. Furthermore, the **generator expression** *expr* may refer to the index position through the **generator variable** *idx_vec*. For example, the WITH-loop

```
with ([1,1] <= iv < [3,4]) : iv[0] + iv[1];
genarray( [3,5], 0)
```

yields the matrix $\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 3 & 4 & 0 \\ 0 & 3 & 4 & 5 & 0 \end{pmatrix}$.

It should be mentioned here, that these WITH-loops constitute a restricted form of the WITH-loops in SAC. Exact definitions of fully-fledged WITH-loops and the formal definitions of transformations on them can be found elsewhere [12,11,10].

3. Running example

We illustrate the combined effect of our three WITH-loop optimizations folding, fusion, and scalarization by a running example. In order to demonstrate complexity and versatility of the individual optimizations without making the example overly complicated, we use a rather artificial function `foo` as shown in Fig. 1. It takes a 9x9-element matrix of complex numbers as an argument and

```
complex[9,9], complex[9,9] foo (complex[9,9] A)
{
  B = take( [5,9], A) +0+ genarray( [4,9], [1.0, 0.0]);
  C = shift( [1,2],
             shift( [-1,-2],
                    A + shift( [1,1],
                               B,
                               [0.0, 0.0]),
                    [0.0, 0.0]),
             [0.0, 0.0]);
  D = take( [9,7], B) +1+ genarray( [9,2], [0.0, 0.0]);
  return( C, D);
}
```

B= C= D=

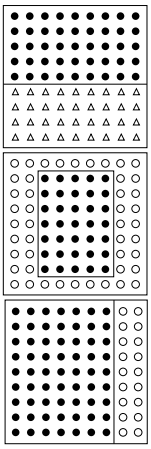


Figure 1. Running example.

yields two such matrices as results.

Both return arrays `C` and `D` are defined in terms of the argument array `A` and an intermediate array `B`. The latter is computed from `A` by taking the first five rows of `A` and by concatenating this 5×9 array with a 4×9 array of complex numbers of value 1. Essentially, the first result array `C` is defined as sums of the corresponding elements of `A` and of `B` where the latter elements are additionally shifted by one index position along both axes. The result of this addition is embedded

in two further shift operations which push zeros into the first and last row as well as into the first and last two columns. For result array D we take the corresponding elements of B for the leftmost seven columns and initialize the remaining two columns to complex zero.

Note here, that all array operations such as **take**, **shift**, **+0+** or + in fact are defined by WITH-loops. Inlining their definitions leads to more then 10 WITH-loops rendering a naive compilation prohibitive. Rather than explaining the entire merging process, we focus on the last few steps that need to be applied after the individual definitions of B, C and D have been merged. Fig. 2 shows a beautified version of that intermediate form. The concatenations with arrays of zeros are expressed

```
complex[9,9], complex[9,9] foo (complex[9,9] A)
{
  B = with ([0,0] <= iv < [5,9]) : A[iv]
      genarray( [9,9], [1.0,0.0]);
  C = with ([1,2] <= iv < [8,7]) : A[iv] + B[iv-1]
      genarray( [9,9], [0.0,0.0]);
  D = with ([0,0] <= iv < [9,7]) : B[iv]
      genarray( [9,9], [0.0,0.0]);
  return( C, D);
}
```

Figure 2. Running example as WITH-loops.

as WITH-loops with default element zero. Similarly, the outer two shift operations in the definition of C have been resolved into a zero default element, and the inner shift operation is expressed as an index offset when accessing the array B.

Before applying any of the three WITH-loop optimizations, all WITH-loops are transformed into a slightly more complex representation that makes the default elements explicit by adding further generators associated with the default expression. Fig. 3 shows the result of this preprocessing step for the second WITH-loop.

```
C = with ([0,0] <= iv < [1,9]) : [0.0,0.0]
         ([1,0] <= iv < [8,2]) : [0.0,0.0]
         ([1,2] <= iv < [8,7]) : A[iv] + B[iv-1]
         ([1,7] <= iv < [8,9]) : [0.0,0.0]
         ([8,0] <= iv < [9,9]) : [0.0,0.0]
    genarray( [9,9]);
```

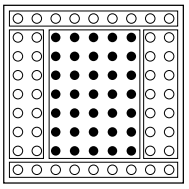
C= 

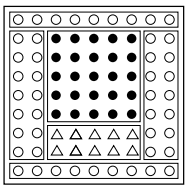
Figure 3. Creating a full partition for the second WITH-loop of the running example.

4. With-loop folding

Our first optimization technique, WITH-loop-folding, addresses vertical compositions of WITH-loops. In the running example introduced in the previous section, we have vertical compositions between the first and the second WITH-loop and again between the first and the third WITH-loop.

Technically spoken, WITH-loop-folding aims at identifying array references within the generator-associated expressions in WITH-loops. If the index expression is an affine function of the WITH-loop's index variable and if the referenced array is itself defined by another WITH-loop, the array reference is replaced by the corresponding element computation. Instead of storing an intermediate result in a temporary data structure and taking the data from there when needed, we forward-substitute the computation of the intermediate value to the place where it is actually needed.

```
complex[9,9], complex[9,9] foo (complex[9,9] A)
{
  C = with ([0,0] <= iv < [1,9]) : [0.0,0.0]
           ([1,0] <= iv < [8,2]) : [0.0,0.0]
           ([1,2] <= iv < [6,7]) : A[iv] + A[iv-1]
           ([1,7] <= iv < [8,9]) : [0.0,0.0]
           ([6,2] <= iv < [8,7]) : A[iv] + [1.0,0.0]
           ([8,0] <= iv < [9,9]) : [0.0,0.0]
      genarray( [9,9]);
  D = with ([0,0] <= iv < [5,7]) : A[iv]
           ([0,7] <= iv < [5,9]) : [0.0,0.0]
           ([5,0] <= iv < [9,7]) : [1.0,0.0]
           ([5,7] <= iv < [9,9]) : [0.0,0.0]
      genarray( [9,9]);
  return( C, d);
}
```

C= 

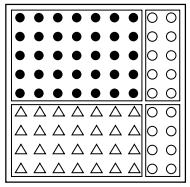
D= 

Figure 4. Running example after WITH-loop-folding.

The challenge of WITH-loop-folding lies in the identification of the correct expression which is to be forward-substituted. Usually, the referenced WITH-loop has multiple generators each being associated with a different expression. Hence, we must decide which of the index sets defined by the generators is actually referenced. To make this decision we must take into account the entire generator sequence of the referenced WITH-loop, the generator of the referencing WITH-loop that is associated with the expression which contains the array reference under consideration, and the affine function defining the index. As demonstrated by the example in Fig. 4, this process generally involves intersection of generators. For example, folding the first WITH-loop into the second one results in splitting the index range of the addition into two separate ones.

5. With-loop fusion

WITH-loop-fusion addresses horizontal compositions of WITH-loops. Horizontal composition is characterized by two a more WITH-loops without data dependences that iterate over the same index space. In our running example the WITH-loops defining the result arrays `C` and `D` form such a horizontal composition. The idea of WITH-loop-fusion is to combine horizontally composed WITH-loops into a more versatile internal representation named multi-operator WITH-loop. The major characteristic of multi-operator WITH-loops is their ability to define multiple array comprehensions and multiple reduction operations as well as mixtures thereof.

Fig. 5 shows the effect of WITH-loop-fusion on the running example. As a consequence of the code transformation both result arrays `C` and `D` are computed in a single sweep. This allows us to share the overhead inflicted by the multi-dimensional loop nest among computing both `C` and `D`.

```
complex[9,9], complex[9,9] foo (complex[9,9] A)
{
  C,D = with ([0,0] <= iv < [1,7]) : [0.0,0.0], A[iv]
             ([0,7] <= iv < [1,9]) : [0.0,0.0], [0.0,0.0]
             ([1,0] <= iv < [5,2]) : [0.0,0.0], A[iv]
             ([5,0] <= iv < [8,2]) : [0.0,0.0], [1.0,0.0]
             ([1,2] <= iv < [5,7]) : A[iv] + A[iv-1], A[iv]
             ([5,2] <= iv < [6,7]) : A[iv] + A[iv-1], [1.0,0.0]
             ([6,2] <= iv < [8,7]) : A[iv] + [1.0,0.0], [1.0,0.0]
             ([1,7] <= iv < [5,9]) : [0.0,0.0], [0.0,0.0]
             ([5,7] <= iv < [8,9]) : [0.0,0.0], [0.0,0.0]
             ([8,0] <= iv < [9,7]) : [0.0,0.0], [1.0,0.0]
             ([8,7] <= iv < [9,9]) : [0.0,0.0], [0.0,0.0]
        genarray( [9,9])
        genarray( [9,9]);
  return( C, D);
}
```

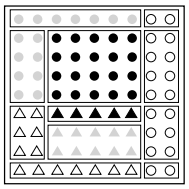


Figure 5. Running example after WITH-loop-fusion.

Furthermore, we change the order of array references at runtime. The intermediate code as shown in Fig. 4 accesses large parts of array A in both WITH-loops. Assuming array sizes typical for numerical computing, elements of A are extremely likely not to reside in cache memory any more when they are needed for execution of the second WITH-loop. With the fused code in Fig. 5 both array references `A[iv]` occur in the same WITH-loop iteration and, hence, the second one always results in a cache hit.

Technically, WITH-loop-fusion requires systematically computing intersections of generators in a way similar to WITH-loop-folding. After identification of suitable WITH-loops, we compute the intersections of all pairs of generators. Whereas this leads to a quadratic increase in the number of generators for the worst case, many of the new generators turn out to be empty in practice as can be seen for our example.

6. With-loop scalarization

So far, we have not paid any attention to the element types of the arrays involved. In SAC, complex numbers are not built-in, but they are defined as vectors of two elements of type double. As a consequence, our 9×9 arrays of complex numbers are in fact three-dimensional arrays of shape `[9,9,2]` and the addition operation on complex numbers in fact is defined by a WITH-loop over vectors of two elements. The idea of WITH-loop-scalarization is to get rid of these nestings of withloops and to transform them into WITH-loops that operate on scalar values. This is achieved by concatenating the bound and shape expressions of the WITH-loops involved and by adjusting the generator variables accordingly. For our example we obtain code equivalent to the code shown in Fig. 6. When comparing this code against the code of Fig. 5, we can observe several benefits. There are no more two-element vectors which results in less memory allocations and deallocations at runtime. Furthermore, the individual values are directly written into the result arrays without any copying from temporary vectors. The fine grain skeletons for the additions of complex numbers have been absorbed within the coarse grain skeleton that constitutes the entire function body now.

```
double[9,9,2], double[9,9,2] foo (double[9,9,2] A)
{
  C,D = with ...
             ([1,2,0] <= iv < [5,7,1]) : A[iv] + A[iv-1], A[iv]
             ([1,2,1] <= iv < [5,7,2]) : A[iv] + A[iv-1], A[iv]
             ...
             ([6,2,0] <= iv < [8,7,1]) : A[iv] + 1.0, 1.0
             ([6,2,1] <= iv < [8,7,2]) : A[iv] + 0.0, 0.0
             ...
        genarray( [9,9,2])
        genarray( [9,9,2]);
  return( C, D);
}
```

Figure 6. Running example after WITH-loop-scalarization.

7. Related work

For imperative array languages, such as various FORTRAN dialects or ZPL [6], optimizations for combining loop constructs typically concentrate on classical fusion techniques [1] that are applied on the level of individual loops rather than entire loop nestings. Forward-substitutions from the body of one loop nesting to another one, as done by WITH-loop-folding, usually are not made due to the lack of a side-effect free setting. Optimizations like WITH-loop-scalarization as well have not been pursued because in an imperative context operational aspects are decoupled from data layout aspects: memory representations of arrays are defined through explicit declaration, not by the operations that incrementally initialize their elements.

In main-stream functional languages such as HASKELL, CLEAN, or ML, separate parts of a program are typically glued together using intermediate data structures other than arrays. However, a considerable amount of research effort went into the development of techniques for their detection and elimination. They are generally referred to as *deforestation* or *fusion* techniques [15,7,8,13]. Although being similar in spirit, they completely differ from our setting as they are based on linked lists while we operate on multidimensional arrays. Array related research in the area of functional programming has mostly focused on achieving reasonable efficiency in general. [2,14,5] discuss issues such as strictness, unboxing, and the aggregate update problem. A variant of deforestation for arrays is described in [4]; it is similar in spirit to WITH-LOOP-FOLDING adapted to the context of HASKELL arrays.

A notable exception from the main-stream of functional programming that puts the emphasis on arrays rather than on lists is SISAL. However, according to [3] the loop optimizations in the context of SISAL are merley restricted to the conventional setting, i.e., to the fusion of individual loops.

8. Conclusions

The design of skeletons for expressing concurrent computations usually faces a conflict between software-engeneering demands and performance issues. The former favor versatile small-grain skeletons that can be successively combined into larger programs, whereas from a performance perspective, coarse grain skeletons are better suitable.

This paper demonstrates in the context SAC how WITH-loop-folding, WITH-loop-fusion, and WITH-loop-scalarization can be used to combine the benefits of both approaches. SAC programs are typically specified as combinations of APL-like array operations that are defined in terms of

fine-grain WITH-loops. After inlining these definitions, the three optimizations, when used jointly, in most cases suffice to merge rather complex compositions of trivial WITH-loops into fewer, more complex WITH-loops that are better suited for an efficent concurrent execution.

References

[1] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, 2001.

[2] S. Anderson and P. Hudak. Compilation of Haskell Array Comprehensions for Scientific Computing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'90), White Plains, New York, USA*, volume 25 of *SIGPLAN Notices*, pages 137–149. ACM Press, 1990.

[3] D.C. Cann. *The Optimizing SISAL Compiler: Version 12.0*. Lawrence Livermore National Laboratory, Livermore, California, 1993. part of the SISAL distribution.

[4] Manuel M.T. Chakravarty and Gabriele Keller. Functional Array Fusion. In *Proceedings of the 6th ACM SIGPLAN International Conference on Functional Programming (ICFP'01), Florence, Italy*, pages 205–216. ACM Press, 2001.

[5] Manuel M.T. Chakravarty and Gabriele Keller. An Approach to Fast Arrays in Haskell. In Johan Jeuring and Simon Peyton Jones, editors, *Summer School and Workshop on Advanced Functional Programming, Oxford, England, UK, 2002*, volume 2638 of *Lecture Notes in Computer Science*, pages 27–58. Springer-Verlag, Berlin, Germany, 2003.

[6] B.L. Chamberlain, S.-E. Choi, C. Lewis, L. Snyder, W.D. Weathersby, and C. Lin. The Case for High-Level Parallel Programming in ZPL. *IEEE Computational Science and Engineering*, 5(3), 1998.

[7] W.N. Chin. Safe Fusion of Functional Expressions II: Further Improvements. *Journal of Functional Programming*, 4(4):515–550, 1994.

[8] A. Gill. *Cheap Deforestation for Non-strict Functional Languages*. PhD thesis, Glasgow University, Glasgow, Scotland, UK, 1996.

[9] C. Grelck. Shared memory multiprocessor support for functional array processing in SAC. *Journal of Functional Programming*, 15(3):353–401, 2005.

[10] C. Grelck, S.-B. Scholz, and K. Trojahner. With-Loop Scalarization: Merging Nested Array Operations. In P. Trinder and G. Michaelson, editors, *Proceedings of the 15th International Workshop on Implementation of Functional Languages (IFL'03), Edinburgh, Scotland, UK, Revised Selected Papers*, volume 3145 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Germany, 2004.

[11] S.-B. Scholz. With-loop-folding in SAC — Condensing Consecutive Array Operations. In C. Clack, T. Davie, and K. Hammond, editors, *Proceedings of the 9th International Workshop on Implementation of Functional Languages (IFL'97), St. Andrews, Scotland, UK, Selected Papers*, volume 1467 of *Lecture Notes in Computer Science*, pages 72–92. Springer-Verlag, Berlin, Germany, 1998.

[12] S.-B. Scholz. Single Assignment C — Efficient Support for High-Level Array Operations in a Functional Setting. *Journal of Functional Programming*, 13(6):1005–1059, 2003.

[13] D. van Arkel, J. van Groningen, and S. Smetsers. Fusion in Practice. In R. Peña and T. Arts, editors, *Proceedings of the 14th International Workshop on Implementation of Functional Languages (IFL'02), Madrid, Spain, Selected Papers*, volume 2670 of *Lecture Notes in Computer Science*, pages 51–67. Springer-Verlag, Berlin, Germany, 2003.

[14] J. van Groningen. The Implementation and Efficiency of Arrays in Clean 1.1. In W. Kluge, editor, *Proceedings of the 8th International Workshop on Implementation of Functional Languages (IFL'96), Bonn, Germany, Selected Papers*, volume 1268 of *Lecture Notes in Computer Science*, pages 105–124. Springer-Verlag, Berlin, Germany, 1997.

[15] P. Wadler. Deforestation: Transforming Programs to Eliminate Trees. *Theoretical Computer Science*, 73(2):231–248, 1990.

Minisymposium

Tools

Scalability of Visualization and Tracing Tools

J. Labarta[a],J. Gimenez[a], E. Martinez[a], P.Gonzalez[a], H. Servat[a], G. Llort[a], X. Aguilar[a]

[a]Barcelona Supercomputing Center, Jordi Girona 1-3, 08034 Barcelona , Spain

Extending the capability of performance tools to deal with the larger and larger machines being deployed is necessary in order to understand their actual behavior and identify how to achieve performance expectations in the frequent case these are not met at a first try. Trace based tools such as Paraver provide extremely powerful and flexible analysis capabilities to identify performance problems not detectable by profile based tools.

Scaling up the usability of trace based tools requires new techniques in both the acquisition and visualization phases. The CEPBA-tools approach distributes the functionalities required to tackle large systems in three different levels. Different acquisition techniques are used in the instrumentation package to control the data captured and maximize the ratio of information to file size. An intermediate level set of tools are used to summarize the generated Paraver traces into smaller traces, with the same format, but where some of the information has been summarized. Examples of filter functionalities at this level include summarization of certain events in periodic software counters and selection of specific time intervals or events. At the final level, different rendering techniques have been introduced in Paraver to visualize traces of many processes while still being able to convey to the analyst the information relevant to identify problems at very coarse level as well as the capabilities to dig down to very detailed levels.

The paper describes in detail the techniques being used along those lines in the CEPBA-tools environment in order to support the analysis of applications run on large systems.

1. Scalability issues in tracing tools

Understanding the behavior of large scale parallel computers is a real need especially in large shared infrastructures offered by HPC centers. With more and more users willing to access such resources it is important to optimize their use, both from the point of view of the individual user aiming at delivering faster results as well as from the point of view of the operator of the resource, interested in maximizing the productivity of all the components in the infrastructure.

The detailed analysis of the behavior of a system requires the ability to capture a large amount of data and process it to deliver information to the analyst. The performance analysis data space can be seen as a three dimensional space, where individual points represent the occurrence of an event relevant for the performance analysis. The three dimensions are time, space (or processors) and event types. Each of them can be very large in itself and the three dimensional space they define can be very densely populated.

Profiling tools [6] essentially summarize all such data at run time within the acquisition process itself and only emit global statistics at the end of the run. Trace based approaches [9][10][8] store the raw data into files that can be analyzed off-line. This has several advantages as the detail is not lost in the summarization process. An iterative analysis loop can be performed where hypotheses can be made and validated. Rather that relying on a predefined set of metrics averaged for a whole run, the focus of the analysis can be dynamically directed to specific metrics and time ranges as it progresses. An intermediate approach [1] does obtain traces, but then precomputes a large set of profile metrics that can be navigated with a graphic interface.

On a scenario of highly parallel platforms, being able to use trace based tools to study the behavior of parallel programs is a challenge with different aspects. First, just storing and handling the data may be a problem. It is easy to face restrictions in storage capacity or in the capability of the analysis tool to process such data with the responsiveness required for the interactive analysis loop. A second issue is how do we present such data to the analyst, specially in typical timeline displays where one line should be used to display for each process the evolution of a given performance index. How to handle and display large amounts of data is an important aspect of the general scalability problem. We nevertheless consider that it is even more important to be able to handle the large dynamic range along the three dimensions earlier presented. A tool should be able to present a very high level view of a whole run and then be able to drill down to a small time interval and a subset of processors where a given microscopic phenomena may have a significant global impact.

The question of performance tools scalability has long been around. The Paradyn project [2][4] was motivated by this concern. This project opened the direction of dynamic instrumentation and iterative automatic search during the program run. In general there is a perception that trace based approaches do face an enormous problem when scaling up to large number of processors. Approaches for structuring the trace data have been proposed [8]. Other work focuses on trace compression mechanisms [6]. These proposals address from the data structure point of view the issue of how to reduce the trace size and at the same time speed up the process of manipulating them either for display or to compute some statistic.

Parallelizing the implementation of the tools themselves is an alternative to speed up their execution or let then use more resources (i.e. file descriptors, memory) than available in a single node. Direct implementations [13] or using general infrastructures [12] to support such parallel implementations are possible.

Our objective is to investigate how far we can go in the use of trace based approaches to analyze the performance of large scale parallel computers. Our position is that even for large numbers of processors, trace based approaches can still be applied, offering the advantages of supporting a detailed analysis and flexible search capabilities. Certainly, blind tracing of a large system is unmanageable. Even if parallelizing the tools or improving the internal data structures will certainly help, it is necessary to explore the direction of intelligent selection of the traced information. In this paper we describe the techniques that have been implemented in the CEPBA-tools environment, centered on the Paraver [8] trace visualization tool. In our environment, the analysis process actually goes through three phases. The first one is the MPI + OpenMP instrumentation package OMPItrace [11]. Current practice in the size of traces generated after an instrumented run is in the order of some GBs. On a standard laptop configuration, Paraver can visualize traces up to 100 MB. To bridge the gap, we have implemented a filtering step where different techniques are used to select or summarize the information. Both the input and output trace are in the same Paraver format.

The structure of the paper is as follows. In section 2 we describe the techniques used in the instrumentation phase. The functionality of the filtering step is described in section 3 and the techniques to display traces of thousands of processors are presented in section 4. Section 5 concludes the paper.

2. Scalability of instrumentation

2.1. Limiting the trace file size

The basic approach supported in many tracing packages is to manually modify the code inserting calls to a tracing library call to start and stop the tracing. Although requiring access to the source code and some understanding of the application structure the mechanism is easy to use either by the code developer or an analyst without a very deep knowledge of the code. This mechanism also

results in a framed traced data, where the start of a phase in the algorithm and its periodicity are easily identified.

In situations where the source code is not available and the structure of the application is not known, simple approaches such as tracing from the beginning of the application till a given trace file size is reached can be quite effective. This mechanism is very easy to use, provides direct control of the amount of data captured and can be tuned to what the post processing or visualization steps can handle. A first drawback appears in applications with a lot of traced activity during the initialization steps. In situations where the behavior of the applications varies along time it may not be possible to reach the actual objective of the analysis.

A desirable functionality is to let the analyst specify what to trace when launching the instrumented run. In our environment this is specified through an environment variable indicating two pairs (function, instance number). Tracing will start when the first function is invoked for the specified time. After that, tracing stops when exiting the specified instance of the second function.

For some applications, the analyst may be interested in tracing a relatively small interval half way through the execution of a very large run even if not having access to the source code. The approach implemented in OMPItrace to support this need is based on a circular buffer. The tracing probes are activated at the beginning of the program and keep storing traced data in the buffer without dumping it to disk. New events will overwrite old ones, but the buffer will always contain the most recent events that fit in it. By sending a signal to all the processes the user forces the dump to disk of the events in the buffer. Correlating the events dumped by different processes is nevertheless tricky. Different paths through the application code and delays in propagation of the dumping signal may result in some of the events in the file generated by one process not having their counterpart in the other. To be able to match events in the dumps of different processes some type of logical synchronization data must be emitted along with the events. In our approach, we rely on collective operations on MPI_COMM_WORLD. The tracing library emits for each such operations its sequence number. The trace merging process searches for such collectives. The first MPI_COMM_WORLD collective appearing on all processes is used as reference for matching events in different processes. Records before that one are discarded. When approaching the end of the local dumps, events in some processes may not have counterpart and are also discarded. Although the mechanism has some limitations (i.e. applications with no global collective calls, codes with high variation in density of MPI calls across processes, point to point communications across collectives, slow propagation of signals), the approach is valid for a large set of codes with very long term behavior variations.

The most desirable situation would be one where the tracing package automatically detects what is the amount of data to be captured. An approach for OpenMP programs described in [3] tries to detect the periodic structure of the application and emits to the tracefile only a few periods of such pattern. The approach has been ported to OMPItrace. As the detection is local to each process, the method has limitations in that we assume an SPMD structure for the program. The approach can be merged with the matching mechanism described for the circular buffer mechanism.

An orthogonal direction to restrict the amount of data in the trace is to limit the set of events captured. The typical analysis of an MPI program would be interested in registering events of entry/exit to user functions and MPI calls as well as hardware counter information at those points. By not emitting some of those data to the file it is possible to restrict the trace size. Depending on the type of analysis, there is no problem in doing so. For example if we know that for a given application the point to point calls are not the actual bottleneck and we are more concerned about the collectives it is possible to only emit such information to the tracefile. If we are interested in the evolution of the load balance for a long run, we may restrict the emitted events to just those on entry and exit to the

user functions that perform the core computation. Even if we do not trace individual MPI calls, the analysis of the code sections encapsulating the communication can still provide a lot of information about communication behavior. Furthermore, even in this case, hardware counter information can unveil great levels of detailed understanding. An example is the Blue Gene/L where the PAPI library used by MPItrace provides hardware counters on the amount of bytes outgoing through each individual link in the 3D torus. Hardware counts at the start and end of communication phases can give detailed information on actual amount of data transferred or level of contention at each link.

2.2. Scalable trace merge

The instrumented run generates one file per process. It is then necessary to merge such dumps into a single trace file, matching send and their corresponding receives and correlating the timestamps.

The merge process is done off line as a batch job. Even if it is desirable for this process not to be very long, the performance requirements for this step are not very high in the typical analysis practice. It must nevertheless match a large number of files and may use a lot of memory. Our previous sequential implementation ran out of file descriptors when tracing above 1000 processes. We have implemented a parallel version that hierarchically merges some individual dumps into partially matched files that are then merged again following a tree structure. Some speedup can be obtained by this parallel merge, but the really important effect is that it enables the merge of traces of large numbers of processors.

3. Trace post processing

For runs with a large number of processors, the above mechanisms can still generate very large tracefiles. To bridge the gap between the GBs generated and tenths of MBs that Paraver can visualize it is necessary to implement filtering mechanism that select or summarize the data. We have implemented several selection and aggregation mechanisms that we describe in the following subsections.We should emphasize that the output of all the filtering steps is again a Paraver trace. This allows for the pipelining of the post processing functionalities. A typical practice will probably undergo several iteration of filtering steps till the final trace showing a specific behavior is obtained. The intermediate output of some of those steps will be visualized and the indications this provides will drive the following filtering steps.

3.1. Selection mechanisms

We have implemented a selection mechanism that extracts from an original trace a subset of the events, states and communication records, emitting them without modification to the output trace, but eliminating all other records. The first basic mechanism selects a time interval by specifying a start and stop time. This can be complemented in the spatial dimension by selecting a subset of the processors. In this case, in order not to loose the information of their interactions with other processes, the communication records that involve one of the selected processes are kept in the output even if the communication counterpart is not selected. Finally, in the states and event dimension it is possible to select a subset of the event types or even a given range of values for those types.

3.2. Software counters

A new mechanism has been implemented in the filter in order to minimize the potentially huge amount of events of a certain type in a trace. The idea draws from current processors where very fine grain events like instruction completion or cache misses are counted and then made available to the monitoring tools at a coarser level of granularity. The software counters mechanism substitutes the individual event instances in one raw trace by a summarized event at the end of a summarization

interval.

Two possible ways of summarization are possible. For events with categorical values indicating for example entry/exit to subroutines (i.e. MPI point to point calls), a count of the number of entries is kept. At the end of the summarization interval an event is emitted indicating the number of invocations during the interval. For events whose associated value is itself a count (i.e. hardware counter events) the individual values are accumulated and a single event of that type is emitted at the end of the interval with the aggregated value.

Two alternatives are possible to determine when to emit the summarized events. A first approach is to do it periodically. This sampling approach is not correlated to the application structure. The relationship between the sampling frequency and the natural frequencies in the application will determine how well its behavior is represented by the software counts. The lower the sampling frequency, the smaller the output trace will be, but if the sampling frequency is too low, everything will be averaged, losing information. For situations with no a priori knowledge of the application behavior it may be necessary to experiment with some sampling periods.

The second alternative is to emit software counter events at points where coarser grain events happen. The typical example would be to accumulate at the exit of user functions counts of how many MPI call occurred since the function was entered.

3.3. Aggregated communication

A similar idea can be applied to the communication data, somewhat preserving the communication structure of the application even if performing an aggregation to reduce the number of events. In its current implementation, the mechanism coalesces several communications between each pair of processors into a single record, with message size equal to the sum of the individual message sizes. For the equivalent message the send is fixed at the send time of the first accumulated message and the receive time is the last receive time of all accumulated messages. One approach to determine which messages to group is to use fixed sampling intervals. All messages going out form a source processor during that interval will be considered for coalescing. Other approach is to group messages whose send time is closer than a specified bound.

This trace compression mechanism introduces an important difference compared to all others described till now. In previous mechanisms, data was either extracted or accumulated, but the data that was generated was accurate. In this case, the generated data is no longer accurate. Even so, it has proven an interesting alternative in the analysis of large runs.

4. Scalability of visualization

4.1. Rendering

Once a trace is available, it is necessary to present to the analyst the data it contains in a way that conveys the maximum possible information about the application behavior [14]. Timelines are a typical way to display such behavior. In this approach, the two dimensional display derives from the three dimensional raw data space by projecting along the event dimension. The events are transformed to some performance metric, function of time that is displayed following some color encoding scheme. Given the limitation in the number of pixels of a display device, the issue arises as to which value represent for each pixel.

A first approach is to aggregate the information for each basic object (thread) according to the logical structure of the application. This can be made if the aggregated metric actually makes sense and represents some property of the set of objects aggregated. The result is that fewer lines have to be represented. For example in an MPI + OpenMP application it may be possible to display the

aggregated MFLOPS for each process. It does not make sense to represent the average identifier of the routine each thread is in, but it may make sense to select the user routine executed by the master thread as representative for the process.

The above approach deals with mechanisms on how to build a given metric for entities in which the program is structured. In Paraver this is part of the semantic module that computes a function of time for each thread, process or the whole application. The approaches we will describe in the following paragraphs are implemented in the Paraver visualization module and are purely related to rendering. The important difference between them lies in their respective quantitative and qualitative nature. The time varying metric computed by the semantic module is accurate and can be used to compute quantitative statistics (counting, averaging, histograms). The objective of the display module is to render such an accurate metric to a small display area. It needs not be accurate and should focus on conveying to the analyst a general perception of relevant aspects of the metric.

A first approach is to display for each pixel the last of the set of values that the sequential display computation assigns to it. This actually corresponds to a periodic sample of the time/space dimension and is thus not correlated to the actual metric to represent. If the sampling frequency is below the natural frequencies in the data the result may be quite misleading. An interesting approach in this situation is to use a random selection of one of the possible values that fall into the pixel area. This is also not correlated to the actual data, but redrawing the display window will report different values and will give the analyst the opportunity to visually inspect the structure of the represented data. Different redraws resulting in structurally similar displays will actually show such structure. Even if different redraws result in very different displays this also conveys to the analyst useful information about the nature of the application behavior.

It is also useful to offer rendering approaches correlated to the data to be represented. A natural one is to average the values that fall into the pixel. This may be useful for quantitative metrics where addition has a physical meaning. It may be less interesting than may be initially though as averaging tends to mask structure rather than highlighting it. Also because it does something that the eye would itself do. Finally, it may make sense to add or average metrics between for example neighboring processors in the actual application problem space, but neighboring processors in such space may not be neighbors in the linear process space of the programming model.

Two simple and very useful rendering methods are to display the maximum or minimum of all values mapping to the pixel. This non linear mechanism does highlight individual processes where some desirable or undesirable behavior appears. We consider that relying on non linear data transformations is extremely useful. In Paraver this is supported by the semantic module building the time varying metric, where it is possible to zero out regions not corresponding to the actual target of the analysis. So if a metric represents for example the MFLOPS inside user function foo and zero elsewhere, it may make sense to display the minimum (not zero) value to get a perception of regions where such routine is performing poorly.

4.2. Focusing on a subset of processes

The global view supported by the above mechanisms is generally used in a first step of the analysis but it is then necessary to focus on a reduced set of processes where a given phenomena shows up. A typical proposal is to provide a scroll bar mechanism for the processor dimension. This was initially implemented in Paraver but proved of little use. Losing the global view picture and having to scroll up and down to search for a given behavior drops the context information the analyst had and makes the search difficult. A two dimension zooming mechanism was then implemented where it is possible to select a region both in time and processes to which the view focuses. Features that support a fairly good navigation through the large timeline representation are the undo/redo

capabilities and the possibility to copy both the time scale and the selected processes to different windows. In this way, some metrics may highlight a given problem on a global view and the analyst can then focus to the specific time/space region where it appears, looking at it with different views and metrics.

It is often the case that the processes involved in a given behavior are not contiguous. When scaling up the system they may not even be near one another in the linear process model numbering of processes. It is necessary to support the selection of an arbitrary subset of processes to display. The GUI can let the analyst tick the desired processes to display. Although this provides the basic mechanism, it is not very scalable form the usability point of view. Even if the selection mechanism is available, the actual problem is to identify which processes to select. Next section describes how this is addressed by Paraver.

4.3. Analysis power and scalability of visualization

When getting deep into an analysis, questions will arise as for example wondering whether there are many processes sending or receiving form process X, how stable is the TLB miss ratio within a given routine or what message sizes appear in the run. The analysis module in Paraver can be used to compute statistics or histograms that give a numerical answer to those questions. The typical situation is then that the analyst wants to have a look at the timeline for those invocations that incur a certain range of TLB misses or MPI calls with message sizes in a given range or just the processors that communicate with processor X.

To satisfy such need, we implemented a mechanism by which selecting a region of the histogram view automatically generates a display window where only time intervals where the metric falls in the selected range are displayed, while all other time intervals are zeroed. Furthermore, only processes with non null entries in that region of the histogram appear in the generated view. The result is a very precise quantitative selection technique to let the analyst focus directly to regions of time and space where a given property is expressed. This mechanism heavily relies on the semantic module of Paraver to build functions of time from the raw events, apply non linear compositions and combine them to derive elaborated metrics.

5. Conclusions

This paper describes the techniques used in the CEPBA-tools environment to scale the applicability of the Paraver trace visualization and analysis tool to systems with up to several thousands of processors. Our objective is to support, for the same analysis a very large dynamic range of granularities in the effects observed.

The techniques have been applied at three different phases: instrumentation, post processing and visualization/analysis. Even if in the initial implementation each phase implemented its own techniques, the experience shows that there is a common underlying set of concepts shared by the three of them (i.e. selection of events or timescales, summarization mechanisms). Based on this, future work will define a common configuration language. We expect this will greatly increase the flexibility and power of the whole environment separating specification of how to process the data from how that is actually carried out.

Regarding data acquisition and handling our approach is based in providing flexible mechanisms to select and summarize the raw data to obtain traces that still contain the relevant information (detail, structure, variance) with much less data. In particular, we have described the software counters technique that has proven very useful to analyze the structure of large runs with many processes.

Once those mechanisms are available, it is possible to apply intelligence in their use to drastically

reduce the data and focus on what is relevant for understanding a given behavior. Such intelligence may be provided by the user/analyst but also automatically. A very relevant direction of ongoing work is the analysis of the large traces in order to automatically generate the filtered traces for the analyst to look at. Example objectives of such analysis are the identification of regions with significant OS perturbation or the detection of the periodic pattern in the application and selection of an appropriate time interval.

Different rendering techniques have been presented. As opposed to a general concern on the issue, we have observed that the limited size of a typical display is not a real problem for analyzing traces of thousands of processors. In particular, non linear rendering has proven very useful. Two other important conclusions regarding the visualization tool are the importance of tightly coupling the quantitative analysis mechanisms and the display selection mechanisms; and the need to further increase the capabilities of the semantic module and quantitative analysis mechanisms.

This work is supported by the Ministry of Science and Technology of Spain (under contract TIN2004-07739-C02-01) the European Union (HPC-Europa project Contract No RII3-CT-2003-506079) and by BSC (Barcelona Supercomputing Center).

References

[1] B. Mohr, F. Wolf: "KOJAK - A Tool Set for Automatic Performance Analysis of Parallel Programs". Procs. of the International Conference on Parallel and Distributed Computing (Euro-Par 2003), Klagenfurt, Austria, August 2003. - (Lecture notes in computer science; 2790). - S. 1301 – 1304.

[2] B. P. Miller, M. Callaghan, J. Cargille, J.K. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, T. Newhall: "The Paradyn Parallel Performance Measurement Tool" IEEE Computer, Nov. 1995

[3] F. Freitag, J. Caubet, J. Labarta: "On the Scalability of Tracing Mechanisms" Euro-Par, pp. 97-104, Paderborn, August 2002.

[4] J. K. Hollingsworth, B. P. Miller, J. Cargille: "DNew Algorithms for Performance Trace Analysis Based on Compressed Complete Call Graphs" V.S.Sunderam el al. (Eds.) ICCS2005, LNCS 3515, pp. 116-123. 2005.

[5] A. Knupfer and W.E. Nagel: "The Dynamic Probe Class Library-An Infrastructure for Developing Instrumentation for Performance Tools" IPDPS 2001, April 2001.

[6] mpiP. http://www.llnl.gov/CASC/mpip/

[7] A. Chan, W. Gropp, and E. Lusk: "Scalable Log Files for Parallel Program Trace Data — DRAFT". Argonne National Laboratory, Argonne, IL 60439, 2000.

[8] Paraver, http://www.cepba.upc.es/paraver

[9] VAMPIR User's Guide, Pallas GmbH, http://www.pallas.de

[10] O. Zaki, E. Lusk, W. Gropp, and D. Swider: "Toward Scalable Performance Visualization with Jumpshot" High-Performance Computing Applications, volume 13, number 2, pages 277-288, 1999.

[11] OMPItrace. http://www.cepba.upc.es/paraver/manual_i.htm

[12] P.C. Roth, D.C. Arnold, B.P. Miller: "MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools"SC2003, Phoenix, AZ, November 2003.

[13] Vampir-ng. http://www.vampir-ng.de/

[14] Steven T. Hackstadt, Allen D. Malony, Bernd Mohr: "Scalable Performance Visualization for Data-Parallel Programs" Proceedings of the Scalable High Performance Computing Conference (SHPCC), Knoxville, TN, May, 1994.

Performance comparison and optimization: Case studies using BenchIT

R. Schöne[a], G. Juckeland[a], W. E. Nagel[a], S. Pflüger[a], R. Wloch[a]

[a]Center for Information Services and High Performance Computing, Technische Universität Dresden, 01062 Dresden, Germany

Quite often, efficient usage of computing resources is a challenge. Performance measurements, comparisons, and the resulting optimizations are a suitable way to accomplish this goal. The BenchIT project provides a framework for performance measurements on UNIX based systems. Our approach combines low requirements on software with a simple interface for measuring kernels and a strict separation of configuration, compiling, measuring, and result evaluation. Unlike most other benchmark systems, the BenchIT environment provides functions on several levels: Measurements on varying problem sizes, graphical presentation of the results, and the capability to automatically compare with other measurements. Those characteristics along with the userfriendly interfaces which allow access to the database, make comparing different algorithms easy when using BenchIT. This paper presents the BenchIT platform and describes first results on selected machines.

Introduction

Performance measurement is complex and frequently discussed in the analysis of computer systems. The diversity and popularity of benchmarks are now part of our ”computer culture”. But how can different systems be compared to each other? Sure, there are standard benchmarks like LINPACK and SPEC - general and built for many systems. However, they have shortcomings. Namely, just the single resulting number, which is in SPEC referred to as the performance of a basic system, and in LINPACK to the achieved GFLOPS. The question is if one single number is enough to specify and categorize a whole computer system. On one hand, having only one value gives the user an abstract overview about the general performance in specific areas. But on the other hand, perhaps one number is not enough. Every user has different requirements regarding a computer system, and the predominance in one specific problem class could be important to him. BenchIT is designed to provide an abstract interface for comparing computer systems. It enables the user to benchmark nearly every possible algorithm on UNIX systems, providing an infrastructure to analyze the results cooperatively. One special test case is the multiplication of dense double precision floating-point matrices which is the subject of further performance considerations in section 3.
Opteron and Itanium based systems form 20 percent of the processors in the actual 25th TOP500 list, as well as 20 percent of the total achieved performance. In this paper we show firstly the influence different compiler flags have on the performance of one single processor and secondly the influence they have on the performance of different BLAS libraries and OpenMP.

1. Measurements with BenchIT

As previously mentioned in other publications ([1], [2]), BenchIT identifies systems with ”LOCALDEF” files, which are written in plain text. They contain a lot of information about the system and are separated into three files. One file holds compiler-and runtime-options like compiler names, flags, libraries, maximum runtime, etc. The next file contains information about the systems architecture which is needed to compare different machines, for instance processor name, clock rate,

memory type.
A tool for semi-automatic generation of this file called ”Architecture Information Database” (A.I.D), is under construction and will be available soon. The last file contains information about how the results are visualized when using gnuplot. These files can be extended and customized in such a way that additional information about the system is added to the database automatically. When trying to compile measurement kernels without existing LOCALDEFS, a routine will try to auto-detect or predefine those settings (See figure 1). After the choosing basic settings like the C-compiler and the processor name, the measurement kernels can be started. The measurement kernels are compiled using a central script, which writes the created executables to a separate folder for binaries. While being compiled, the kernels store the systems environment variables inside the binary. When being executed by the runscript, they restore the original environment from the compile time. The advantages are obvious: Having a strict separation of compilation and execution, it is possible to start the processes on batch systems with exactly the same environment as when the kernel was compiled. It simply works better with cross-compilers and compiled versions still available for measurements later.
The compiled kernels can be measured on several code compatible systems by resetting the differing variables. Furthermore, the precision of measurements can be increased by changing the number of measurement cycles and by using performance counter libraries like PAPI [3] or PCL [4]. Skeletons for measurement-kernels are also available for use of these libraries.
When measurements finished, results thus obtained can be used to create different graphic files locally or to upload them to the website[1], where they can be compared with other peoples results. It is possible to share them with other people and groups.
Each step can also be done with a GUI, which helps novice users but also accelerates the work of professionals. It helps filling out the LOCALDEFS with a graphical editor, allows editing and building new measurement kernels with a build in IDE[2], starting them by providing a graphical interface for the kernels and plotting the results.
With the GUI, it is possible to run jobs on other systems using standard tools like ssh and tar. It is also possible to run the GUI as a simple data collector, so that Windows users can still benchmark their UNIX servers. Results from the database can also be obtained and compared with local ones. Figure 2 shows the GUI as it edits a kernel while two remote-jobs are running. For full color screenshots please visit our homepage. The entire work flow with BenchIT is shown in figure 3.

2. Observed System Architectures

The environments for the discussed performance measurements are found in Opteron and Itanium 2 cluster. The system characteristics are shown in table 1. Parallelization in an Opteron cluster occurs at three different levels. The first one is hardware based, typically pipelining or
superscalarity. This level is handled internally by the processor and is therefore not discussed here. The next level is also processed internally. However, it is prescribed by the programmer or the compiler and uses an executable code. Examples are the Bit Level Parallelism and the usage of SIMD extensions. These extensions are not only supported by compiler libraries, but also used efficiently for code optimization. The highest level of parallelization is the utilization of several processors for solving the problem. For this purpose we studied optimized libraries with and without the usage of OpenMP. More information about the AMD64 family can be found as reported in the ZIH ([5]).

[1] `www.benchit.org`
[2] Integrated Development Environment

Table 1
Observed Systems

Architecture	IA-64	x86-64
Processor	Intel Itanium 2 Madison	AMD Opteron 248
Clockrate	1.4 GHz	2.2 GHz
RAM	4 GB	$\geq$ 2GB/node
Operating System	SuSE Enterprise Server 9	SuSE Enterprise Server 9
Kernel	2.6.5	2.6.5
available Compilers	GNU 4.0.1, Intel 9.0	GNU 4.0.1, Intel 9.0 Intel 8.1, Portland Group 6.0

Parallelization in the Itanium 2 Cluster starts on the same levels, but does not support SIMD-Extensions in IA-64 mode.

3. Performance Results

First, the maximum achievable performance for the measured algorithm, a matrix multiplication of dense matrices, on the introduced systems is checked. Therefore, the algorithm is measured with several optimized libraries. Then, the performance received from compiler generated code is compared against these results. The AMD Opteron has a peak performance of about 4 GFLOPS using ACML[3], which is close to its theoretical peak-performance. There is, however, no measured performance gain in the usage of packed-SSE2-instructions on AMD Opteron systems in a simple SSE2-implementation of the matrix multiplication. The Intel Itanium 2 processor reaches a peak performance of about 5.4 GFLOPS when using the Intel MKL[4]. All of the following results are gained with optimization-level -O3, unless otherwise noted.

3.1. Library Results

On AMD Opteron, ACML and ATLAS[5] show nearly the same performance approaching 4 GFLOPS whilst Intels MKL reaches a maximum of 1 GFLOPS. Also, the MKL breaks down abruptly at problem size 1290 and only recovers to a rate of 350 MFLOPS, as can be seen in figure 4. When running ATLAS on two processors, an acceleration of more than 1 is achieved for problem sizes larger then 200. Beyond that, it starts to speedup to 1.8. While the sequential work with ACML shows a continuous function for the different problem sizes, the parallel work shows a sawtooth-like behavior. On Intel Itanium 2, ATLAS reaches about 5 GFLOPS for large matrices. This value is reached for the first time durable on problem size 1600. Intels MKL shows a more continuous behavior. It reaches its maximum performance of 5.5 GFLOPS at problem size 300.

3.2. FORTRAN Results

FORTRAN is and has been optimized in numerical calculations. From the beginning it has supported programmers with data types and operations, which are still missing in other programming languages. Especially for scientific and numerical routines, FORTRAN is indispensable.
When comparing different commercial compilers on AMD Opteron, the peak performances between

[3] AMD Core Math Library
[4] Math Kernel Library
[5] Automatically Tuned Linear Algebra Software

them are nearly the same level. Both Intel 9.0 and Portland 6.0 show about 1.2 GFLOPS, which can be seen in figure 6. With a problem size between 100 and 300, Intel shows a better behavior, with different tableaus. The best performing permutation reaches a level of 1.1 GFLOPS. Pgfort also produces tableaus, but only for non-performing permutations. The GNU-Compiler only reaches a maximum of 970 MFLOPS out of the L1 Cache. When the data structures do not fit in any caches, the performance of all 3 compilers are between 20 and 420 MFLOPS.
Interestingly, the Intel Compiler 9.0 reaches a maximum of 1.2 GFLOPS, whereas its predecessor, icc 8.1, achieves 1.5 GFLOPS when the data fits in the L1 Cache. So a speedup of 0.8 can be seen in this generation step. Larger matrices that do not fit in the L1 Cache show only small performance differences, as seen in figure 7. Interesting behaviour is also seen in the Portland Compiler. While the most performing permutation under GNU and Intel Compilers is jki, the ikj-permutation exceeds the others when the datastructures fits in the Cache. When the data structures are larger, the results of the different compilers converge. This is also the reason why the best permutations do not show a tableau for problem sizes which run within the L2 Cache.
Running the same routine on an Intel Itanium 2, the Intel Compiler recognizes the matrix multiplication and replaces it with an optimized implementation. This allows a maximum performance of 5.1 GFLOPS and a sustained performance of 4.5 GFLOPS. While the Intel Compiler shows great results, the GNU compiler does not even reach 100 MFLOPS.

3.3. C Results

C is known for efficiency. Although it is also used for writing applications, it is the most popular programming language for writing system software. When compiling the matrix multiplication on the AMD Opteron with different compilers and a constant flag `-O3`, the Intel C-Compiler dominates the others with a maximal performance of 1.2 GFLOPS. The performance of the different permutations is as expected: ikj is the fastest (kji the slowest) with 1200(670) MFLOPS, when the data fits in the L1-Cache, and 420(25) MFLOPS, when the matrices are too large to fit any Caches. It is repeatedly shown that the predecessor offers a better performance for small matrices which fit in the L1-Cache. The Intel Compiler 8.1 reaches 1.3 GFLOPS.
The executable created by icc is a bit faster than the one generated by pgcc, which reaches a maximum performance of 1.15 GFLOPS. According to the FORTRAN results, when compiling with pgcc, the permutation kji runs the fastest, as long as the data fits into the L1-cache. As expected, however, when the size of the problem increases the permutation efficiency drops.
The GNU Compiler reaches a maximum performance of 960 MFLOPS. This value is achieved by 3 permutations and is relatively similar to the pgcc results for ikj and kij. When changing the the iccs compiler flags from `-O3` to `-O2` or `-Os`[6], the performance level stays at same. Using the gcc-option for 32-bit-code (`-m32`) results in a maximum performance of 550 MFLOPS. But by forcing the cpu to an AMD K8 (march=k8), the performance for large problem sizes remains above the level from using only `-O3`.
With Intel Itanium 2, the GNU Compiler does not reach a satisfactory result, with a performance peak of 190 MFLOPS. This peak, however, is twice as good as the result for the GNU FORTRAN Compiler. The Intel Compiler reaches, as expected, a higher level with a maximum of 650 MFLOPS, which is more then 3 times the gcc performance. When guaranteeing no overlapping data structures with flag `-fno-alias`, the performance increases rapidly with icc. The code is optimized: a maximum performance of 2.2 GFLOPS is reached. Sustained, it is still around 1.2 GFLOPS for all permutations except kij, which shows its normal behavior. Overall, problem sizes which are

[6] enable speed optimizations, but disable some optimizations which increases code size for small speed benefit

multiples of 16 show the best performance for the optimized permutations, as seen in figure 5.

4. Conclusion and further work

BenchIT is a powerful and flexible tool which allows performance measurements of POSIX.1 conform computers. It supports users to write performance measuring kernels as well as compile and run kernels in different environments. Using BenchIT, one can easily plot results to share them among users worldwide. This benchmarking suite, also userfriendly and flexible to use, allows easy system characteristic comparisons, such as the type of RAM, the size of caches, or the processors clock rate.
Clear conventions, extendable configurations, and its independence from specific platforms allow this project to advance into many directions. Further development in this area will offer new possibilities to measure and compare different systems and environments. New batch-environments will be specified, new measurement kernels developed and, of course, the supportive tools will advance even further.

References

[1] JUCKELAND, Guido; BÖRNER, Stefan; KLUGE, Michael; KÖLLING, Sebastian; NAGEL, Wolfgang E.; PFLÜGER, Stefan; RÖDING, Heike; SEIDL, Stefan; WILLIAM, Thomas; WLOCH, Robert:
BenchIT - Performance Measurements and Comparison for Scientific Applications.
In: JOUBERT, Gerhard R. (Hrsg.) ; NAGEL, Wolfgang E. (Hrsg.) ; PETERS, F. J. (Hrsg.) ; WALTER, W. V. (Hrsg.): *PARCO* Bd. 13, Elsevier, 2003. –
ISBN 0–444–51689–1, S. 501–508

[2] JUCKELAND, Guido; KLUGE, Michael; NAGEL, Wolfgang E.; PFLÜGER, Stefan:
Performance Analysis with BenchIT: Portable, Flexible, Easy to Use.
In: *QEST*, IEEE Computer Society, 2004. –
ISBN 0–7695–2185–1, S. 320–321

[3] INNOVATIVE COMPUTING LABORATORY, UNIVERSITY OF TENNESSEE.
PAPI Website.
`http://icl.cs.utk.edu/papi`

[4] ZIEGLER, Heinz; MOHR, Bernd
CENTRAL INSTITUTE FOR APPLIED MATHEMATICS (ZAM) AT THE RESEARCH CENTER JUELICH, GERMANY.
PCL Website.
`http://www.fz-juelich.de/zam/PCL/`

[5] JUCKELAND, Guido
CENTER FOR INFORMATIONSERVICES AND HIGHPERFORMANCECOMPUTING (ZIH) AT THE TECHNISCHE UNIVERSITÄT DRESDEN, GERMANY.
The AMD64 Chip Family: Concepts and First Measurements .
ZHR-R-0403

Figure 1. Autodetection of different environment settings

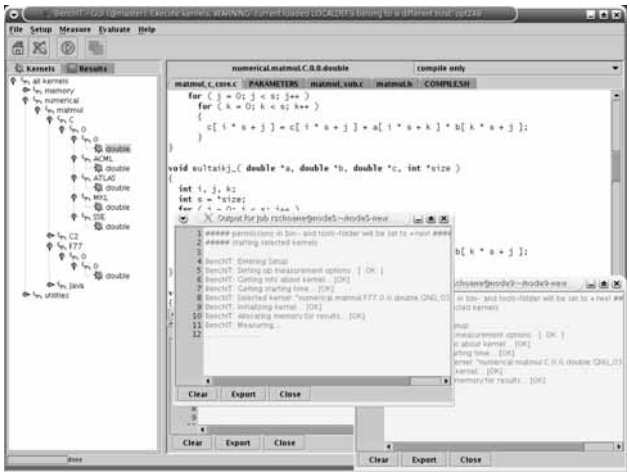


Figure 2. GUI

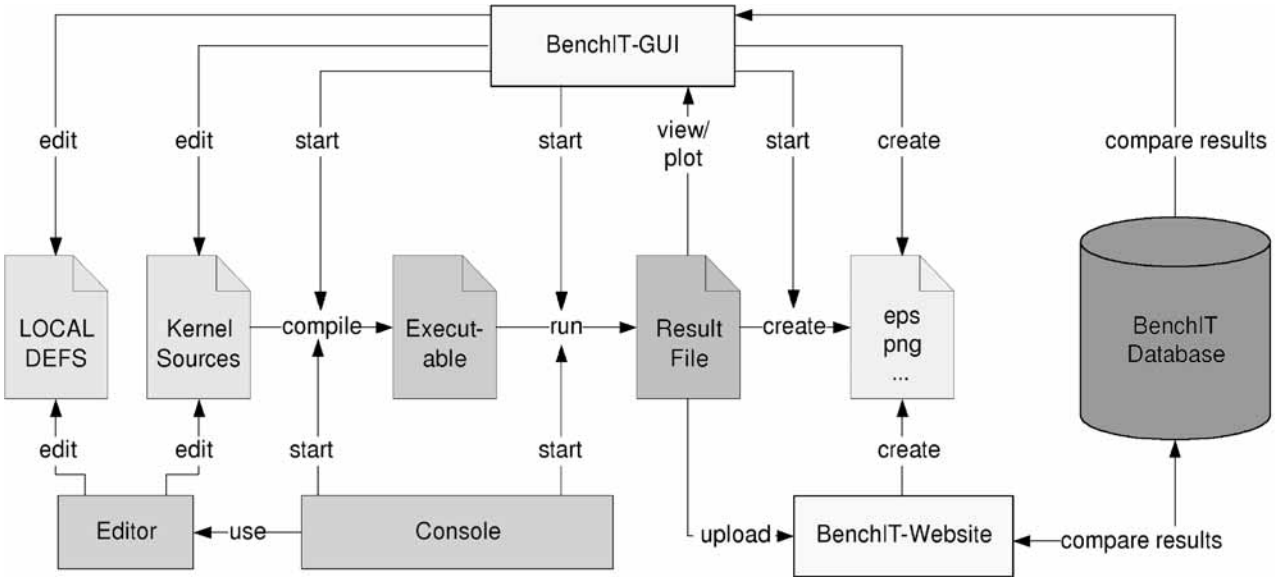


Figure 3. BenchIT - functional overview

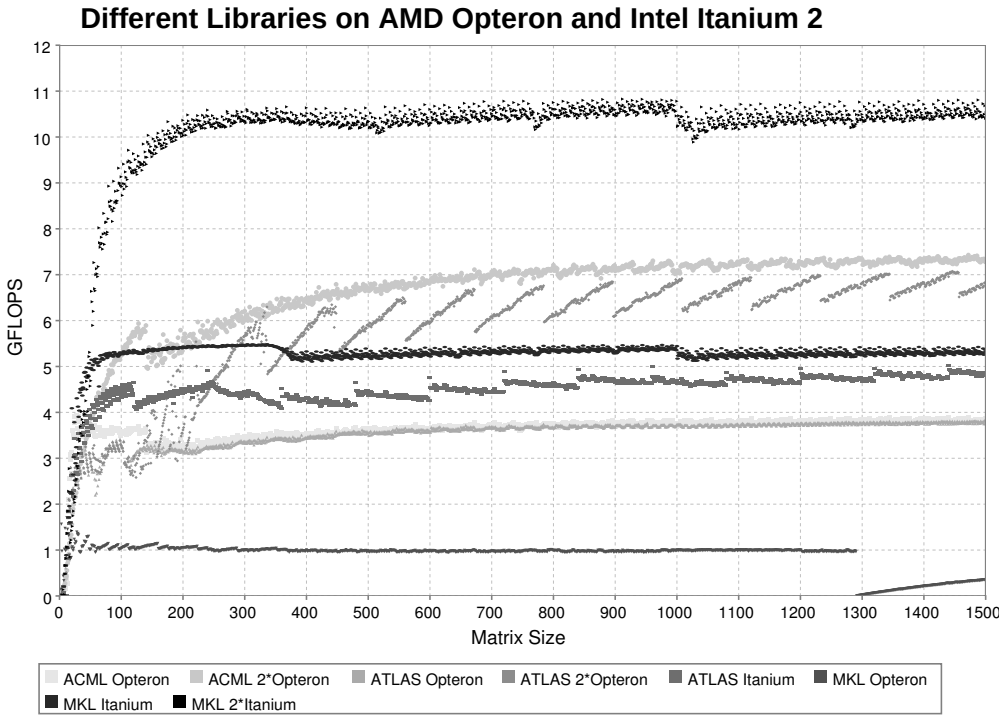


Figure 4. DGEMM with MKL, ATLAS and ACML on AMD Opteron and Intel Itanium 2

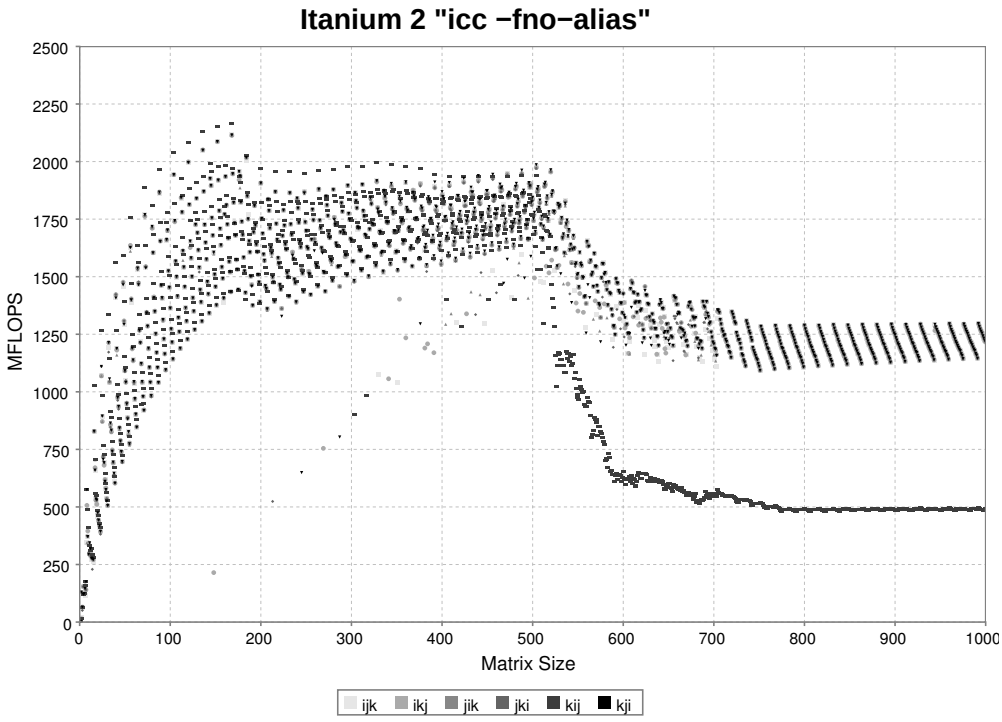


Figure 5. Matrix multiplication with ”icc -fno-alias” on Intel Itanium 2

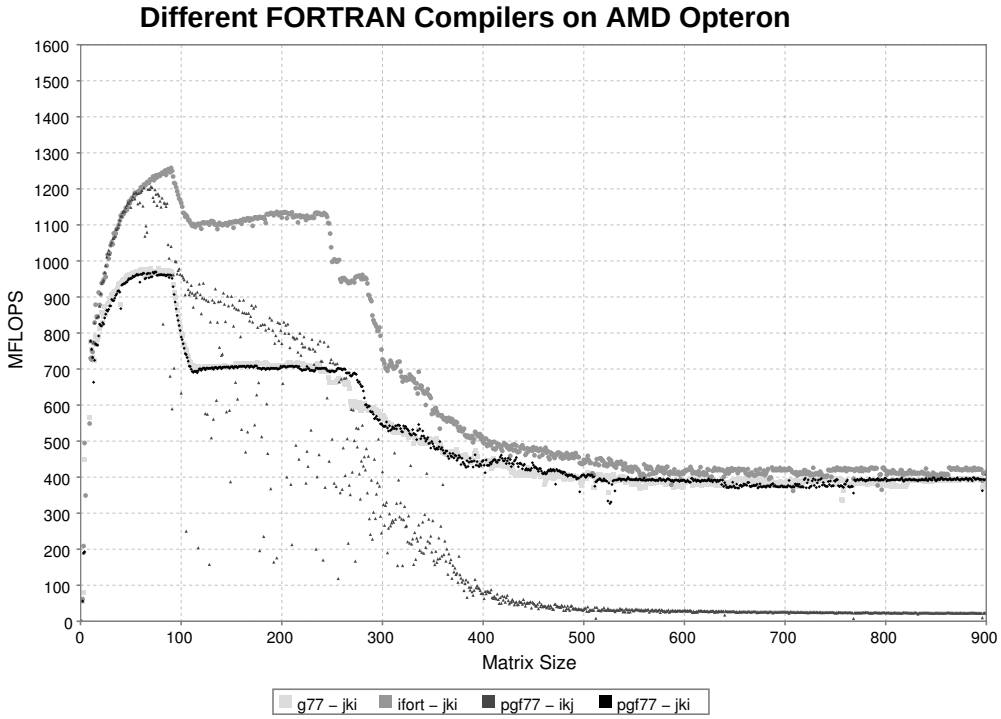


Figure 6. Matrix multiplication with different compilers on AMD Opteron

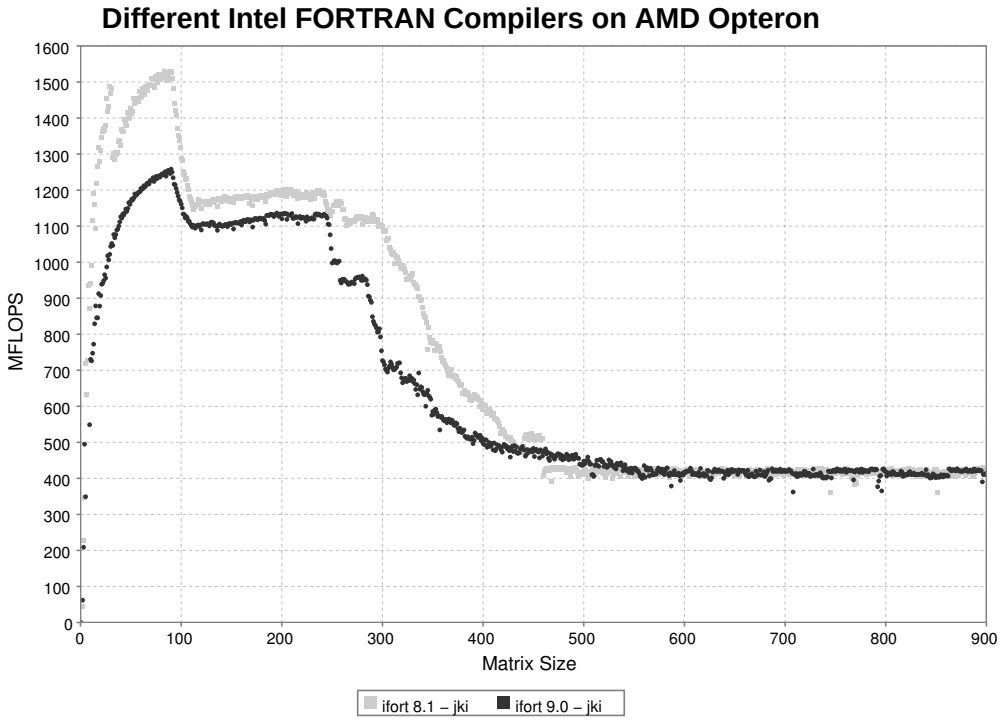


Figure 7. Matrix multiplication on AMD Opteron with different versions of ifort

Performance Analysis of One-sided Communication Mechanisms

B. Mohr[a], A. Kühnal[a], M.-A. Hermanns[a], F. Wolf[a]

[a]Forschungszentrum Jülich, ZAM, 52425 Jülich, Germany

Abstract. Performance analysis of parallel programs requires information about the dynamic behavior of all participating processes. The dynamic behavior can be modeled as a stream or trace of events. The events are chosen in such a way that they represent important aspects in the execution of the application on a level of abstraction suitable for the analysis task. Based on this idea, the KOJAK toolkit for performance analysis records and analyzes the activities of MPI-1 point-to-point and collective communication.

This paper describes the integration of performance measurement and analysis methods for remote memory access (RMA) or one-sided communication into the KOJAK toolkit, in particular for the MPI-2 and SHMEM interfaces. We introduce the underlying event model used to represent the dynamic behavior of RMA operations and show that our model reflects the relationships between communication and synchronization more accurately than existing models. Then, we present event patterns which are used by KOJAK to locate inefficient situations in a program's dynamic remote memory access behavior.

1. Introduction

Remote memory access (RMA) describes the ability of a process to directly access a part of the memory of a remote process, without explicit participation of the remote process in the data transfer. As all parameters for the data transfer are determined by one process, it is also called *one-sided* or *single-sided* communication. On platforms with special hardware providing efficient RMA support, one-sided communication is often made available to the programmer in the form of libraries, for example SHMEM (Cray/SGI) or LAPI (IBM). However, these libraries are typically platform- or at least vendor-specific.

This is one of the reasons why the MPI forum decided to define a portable one-sided communication interface as part of MPI-2. The Message Passing Interface (MPI) was defined by a group of vendors, government laboratories and universities in 1994 as a community standard [1]. This has become known as MPI-1. In 1997, a second version of the interface (MPI-2) was defined, which added support for parallel I/O, dynamic process creation, and one-sided communication [2].

Until recently there was only rare usage of RMA features in scientific applications and, therefore, the demand for performance tools in this area was limited. As more and more programmers adopt the new features to improve the performance of their codes, this is expected to change. For example, NASA researchers report a 39% improvement in throughput after replacing MPI-1 non-blocking with MPI-2 one-sided communication in a global atmosphere simulation program [3].

KOJAK, our toolkit for automatic performance analysis [9], is jointly developed by the Central Institute for Applied Mathematics of the Research Centre Jülich and by the Innovative Computing Laboratory of the University of Tennessee. It is able to instrument and analyze OpenMP constructs and MPI-1 calls. In this paper we report on the integration of performance analysis methods for one-sided communication into the existing toolkit. We introduce an extension to our event model that realistically represents the dynamic behavior of MPI-2 RMA operations in the event stream. We show that our model reflects the relationships between communication and synchronization more

accurately than existing models. The model is general enough to also cover alternate, but simpler, RMA interfaces. In addition, we present KOJAK's new performance properties used to analyze MPI-2 and SHMEM parallel programs. *Performance properties* are event patterns which are used by KOJAK to locate inefficient situations in a program's dynamic behavior.

In our new prototype implementation, we added support for measurement and analysis of parallel programs using MPI-2 and SHMEM one-sided communication and synchronization. We are also able to handle Co-Array Fortran programs [8], a small extension to Fortran 95 that provides a simple, explicit notation for one-sided communication and synchronization, expressed in a natural Fortran-like syntax. Details of this work can be found in [10].

The remainder of the paper is organized as follows: First, we summarize related work in Section 2. Section 3 gives a short description of the MPI-2 RMA communication and synchronization functions. In Section 4, we present our event model, which realistically represents the dynamic behavior of RMA operations. KOJAK's hierarchy of performance properties for the analysis of RMA communication and synchronization is described in Section 5. Finally, we present conclusions and future work in Section 6.

2. Related Work

Currently, there are only very few tools which support the measurement and analysis of one-sided communication and synchronization on a wide range of platforms. The well-known Paradyn tool which performs an automatic on-line bottleneck search, was recently extended to support several major features of MPI-2 [4]. For RMA analysis, it collects basic, process-local, statistical data (i.e., transfer counts and execution time spent in RMA functions). It does not take inter-process relationships into account nor does it provide detailed trace data. Also, it does not support analysis of SHMEM programs. The very portable TAU performance analysis tool environment [5] supports profiling and tracing of MPI-2 and SHMEM one-sided communication. However, it only monitors the entry and exit of the RMA functions; it does not provide RMA transfer statistics nor are the transfers recorded in tracing mode. The commercial Intel Trace Collector tool (formerly known as VampirTrace) [6] records MPI execution traces. When used with MPI-2, only a subset of the RMA functions are traced. It also traces the actual RMA transfers, but misrepresents their semantics, as defined by MPI-2. Finally, it does not record the collective nature of MPI-2 window functions. Besides these there are also some non-portable vendor tools with similar limitations.

3. MPI-2 One-sided Communication

The interface for RMA operations defined by MPI-2 differs from the vendor-specific APIs in many respects. This is to ensure that it can be efficiently implemented on a wide variety of computing platforms even if a platform does not provide any direct hardware support for RMA. The design behind the MPI-2 RMA API specification is similar to that of weakly coherent memory systems: correct ordering of memory accesses has to be specified by the user with explicit synchronization calls; for efficiency, the implementation can delay communication operations until the synchronization calls occur.

MPI does not allow access to arbitrary memory locations with RMA operations, but only to designated parts of a process's memory, the so-called *windows*. Windows must be explicitly initialized (with a call to `MPI_Win_create`) and released (with `MPI_Win_free`) by all processes that either provide memory or want to access this memory. These calls are *collective* between all participating partners and include an internal barrier operation. MPI denotes by *origin* the process that performs

an RMA read or write operation, and by *target* the process in which the memory is accessed.

There are three RMA communication calls in MPI: `MPI_Put` transfers data from the caller's memory to the target memory (*remote write*); `MPI_Get` transfers data from the target to the origin (*remote read*); and `MPI_Accumulate` updates locations in the target memory, for example, by replacing them with sums or products of the local and remote data values (*remote update*). These operations are *nonblocking*: the call initiates the transfer, but the transfer may continue after the call returns. The transfer is completed, both at the origin and the target, only when a subsequent synchronization call is issued by the caller on the involved window object. Only then are the transferred values (and the associated communication buffers) available to the user code. RMA communication falls in two categories: *active target* and *passive target* communication. In both modes, the parameters of the data transfer are specified only at the origin, however in active mode, both origin and target processes have to participate in the synchronization of the RMA accesses. Only in passive mode is the communication and synchronization completely one-sided.

RMA accesses to locations inside a specific window must occur only within an *access epoch* for this window. Such an access epoch starts with an RMA synchronization call, is followed by any number of remote read, write, or update operations in this window, and finally completes with another (matching) synchronization call. Additionally, in active target communication, a target window can only be accessed within an *exposure epoch*. There is a one-to-one mapping between access epochs on origin processes and exposure epochs on target processes. Distinct epochs for a window on the same process must be disjoint. However, epochs pertaining to different windows may overlap.

MPI provides three RMA synchronization mechanisms:

Fences: The `MPI_Win_fence` collective synchronization call is used for active target communication. An access epoch on an origin process or an exposure epoch on a target process are started and completed by such a call.

General Active Target Synchronization (GATS)**:** Here synchronization is minimized: only pairs of communicating processes synchronize, and they do so only when needed to correctly order accesses to a window with respect to local accesses to that window. An access epoch is started at an origin process by `MPI_Win_start` and is terminated by a call to `MPI_Win_complete`. The start call specifies the group of targets for that epoch. An exposure epoch is started at a target process by `MPI_Win_post` and is completed by `MPI_Win_wait` or `MPI_Win_test`. The post call specifies the group of origin processes for that epoch.

Locks: The `MPI_Win_lock` and `MPI_Win_unlock` calls provide shared and exclusive locks. They are used for passive target communication.

In all cases, data read or written is only accessible from user code after the "closing" synchronization call. It is implementation-defined whether some of the described calls are blocking or nonblocking; for example, in contrast to other shared memory programming paradigms, the lock call does not need to be blocking. For a complete description of MPI-2 RMA communication see [2].

4. An Event Model for One-sided Communication

In this section, we summarize the event types and event models used by KOJAK to realistically represent the behavior of MPI-2 as well as Co-Array Fortran and vendor-specific RMA operations. For a more detailed description of the models and the implementation of KOJAK's monitoring components for one-sided communications see [11]. For a complete description of KOJAK's event types for MPI-1 and OpenMP and of its analysis features see [7, 9].

KOJAK's Event Types for Modeling One-sided Communication

Abstraction	Event type	Type specific attributes
Start / end / origin of RMA one-sided transfers	PUT_1TS	window id, rma id, length, dest loc
	PUT_1TE	window id, rma id, length, src loc
	GET_1TO	window id, rma id
	GET_1TS	window id, rma id, length, dest loc
	GET_1TE	window id, rma id, length, src loc
Leaving MPI GATS function	MPIWEXIT	window id, region id, group id
Leaving MPI collective RMA function	MPIWCEXIT	window id, region id, comm id
Locking / unlocking a MPI window	WLOCK	window id, lock loc, type
	WUNLOCK	window id, lock loc

For the analysis of parallel scientific applications, events that capture the most important aspects of the parallel programming paradigm used (e.g., MPI or OpenMP) and the entering and leaving of surrounding user regions (e.g., functions or loops) are typically defined. In the case of collective MPI functions and OpenMP constructs, instead of "normal" EXIT events, special collective events are used to capture the attributes of the collective operation (e.g., the communicator). MPI-1 point-to-point messages are modeled as pairs of SEND and RECV events. In OpenMP applications, FORK and JOIN events mark the start and end of parallel regions and ALOCK and RLOCK events mark the acquisition and release of locks.

In order to be able to also analyze RMA operations, we defined the new event types shown in Table 1. Start and end of RMA one-sided transfers are marked with PUT_1TS and PUT_1TE (for remote writes and updates) or with GET_1TS and GET_1TE (for remote reads). For these events, we collect the source and destination and the amount of data transferred, as well as a unique RMA operation identifier which allows an easier mapping of #_1TE to the corresponding #_1TS events in the analysis stage later on. For all MPI RMA communication and synchronization operations we also collect an identifier for the window on which the operation was performed. Exits of MPI-2 functions related to general active target synchronization (GATS) are marked with a MPIWEXIT event which also captures the groups of origin or target processors. For collective MPI-2 RMA functions we use a MPIWCEXIT event and record the communicator which defines the group of processes which participate in the collective operation. Finally, MPI window lock and unlock operations are marked with WLOCK and WUNLOCK events.

Based on these event types and their attributes, we introduced two event models for describing the dynamic behavior of RMA operations. For each model, we describe its basic features and analyze its strengths and weaknesses.

4.1. Basic Model

In the first and simpler model, it is assumed that the RMA communication functions have a blocking behavior, that is, the data transfer is completed before the function is finished. Also, RMA synchronization functions are treated as if they were independent of the communication functions.

The invocations of RMA communication and synchronization functions are modeled with ENTER and EXIT events. To model the actual RMA transfer, the transfer-start event is associated with the source process immediately after the beginning of the corresponding communication function. Accordingly, the end event is associated with the destination process shortly before the exit of the (same) function.

The advantage of this model is a straight-forward implementation because events and their at-

tributes can be recorded at exactly the place and time where they are supposed to appear in the model. We use this model for analyzing SHMEM and Co-Array Fortran programs. However, for MPI-2, this model is not sufficient because it ignores the necessary synchronization, as described in Section 3. Since the end-of-transfer event is placed before the end of the communication function, the transfers are recorded as completed even when this is not true, for example, in the case of a nonblocking implementation. Even if the implementation is blocking, it still does not reflect the user-visible behavior. Therefore, in case of MPI-2, we use an extended model, which is described in the next subsection.

4.2. Extended Model

The *extended model* observes the MPI-2 synchronization semantics and, therefore, better reflects the user-visible behavior of MPI-2 RMA operations. The end of fences and GATS calls is now modeled with MPIWCEXIT or MPIWEXIT respectively in order to capture their collective nature. The transfer-start event is still located in the source process immediately after the beginning of the corresponding communication function (as it is in the basic model). However, the transfer-end event is now placed in the destination process shortly before the exit of the RMA synchronization function which completes the transfer according to the MPI-2 standard rules. The extended model removes all disadvantages of the basic model, and for most MPI-2 implementations (which have a non-blocking behavior), it is even closer to reality. However, the model is more complex and the events can no longer be recorded at the location where they appear in the model. Therefore, a complex post-processing of the collected event trace becomes necessary.

5. Performance Properties of One-sided Communication and Synchronization

In this section we describe the analysis KOJAK is performing for execution traces of applications using one-sided communication. KOJAK's analyzer, named EXPERT [12], attempts to prove *performance properties* for one execution of a parallel application and to quantify them according to their influence on the performance. A performance property characterizes a class of performance behavior and is specified in terms of a *compound event*, which the analyzer tries to detect in an event trace. A compound event is a set of events matching a specific execution pattern, whose constituents are connected by relationships and constraints. For each property, EXPERT calculates a *severity* measure indicating the fraction of the total execution time spent on that property and, thus, allows the correlation of different properties in a single view.

EXPERT organizes the performance properties in a hierarchy. The upper levels of the hierarchy (i.e., those that are closer to the root) correspond to more general behavioral aspects such as time spent in MPI functions. The deeper levels correspond to more specific situations such as time lost due to blocking communication. Figure 2 shows the hierarchy of predefined performance properties based on time measurements that are supported by the current version (2.2) of EXPERT. It also supports hierarchical analysis based on hardware counter metrics [14].

The set of performance properties consists of two types. The first type, which constitutes the upper layers of the hierarchy and which is indicated by white boxes, is based on summary information involving, for example, the total execution times of special MPI routines, which could also be provided by a profiling tool. However, the second type, which constitutes the lower layers of the hierarchy and which is indicated by gray boxes, involves idle times that can only be determined by comparing the chronological relation between individual events. A detailed description of the properties for MPI-1 and OpenMP can be found in [9]. In the following, the new set of properties for MPI-2 RMA and SHMEM are presented. An exact mathematical definition of the new properties can be found in [13].

5.1. MPI-2 RMA Performance Properties

The performance properties for MPI-2 RMA are of course part of EXPERT's overall hierarchy for MPI (see Figure 2, upper part) under the categories *Communication* and *Synchronization*. The upper part of the *RMA Synchronization* property tree captures how much execution time is spent on the different MPI-2 synchronization methods *Fence*, *Locks*, and *Active Target*, and on *Window Management* functions which also contain synchronization because of their collective nature.

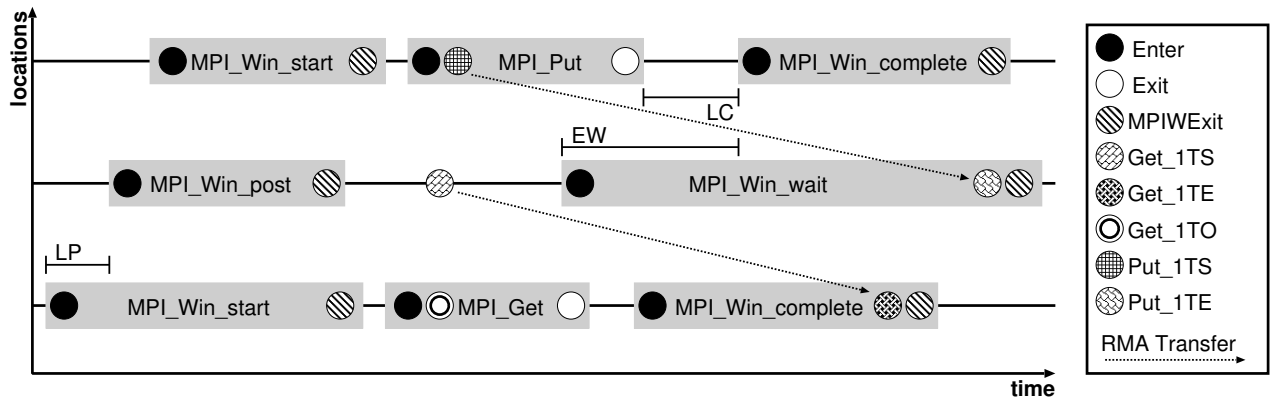


Figure 1. Execution pattern for MPI-2 general active target synchronization.

The first three compound events *Wait at Create*, *Wait at Free*, and *Wait at Fence* simply cover the time spent on waiting in front of these collective operations. The severity for each process is defined by the time from starting the operation until the last participating process arrives. The remaining compound events are related to communication scenarios where general active target synchronization is used. A typical execution sequence is shown in Figure 1. The performance property *Early Wait* is associated with processes providing access to an RMA window (e.g., the middle process in Figure 1) and describes the wasted time waiting for the accesses to complete. The severity is the time spent in `MPI_Win_wait` until the last participating process indicates the end of the accesses by a call to `MPI_Win_complete` (indicated by the interval `EW` in the figure). The subproperty *Late Complete* is associated with the subinterval of this waiting time from the end of the last RMA transfer operation (e.g., the put operation by the upper process in the figure) to the start of the last `MPI_Win_complete` call (marked with `LC`). The property *Late Post* describes the situation where a call to `MPI_Win_start` blocks because the corresponding exposure epoch has not started yet (which is initiated by a `MPI_Win_post` call). As severity we use here the time spent blocked until the start of the post call (see interval `LP` in the figure). On MPI implementations where these synchronization calls are non-blocking, in a similar situation the first RMA transfer call would block. In this case, we call the property *Early Transfer*. It is a subproperty of RMA *Communication* as the blocking occurs during a communication call.

5.2. SHMEM Performance Properties

The performance properties for the SHMEM programming paradigm are modeled after the corresponding properties for MPI (see Figure 2, lower part). The higher level of the SHMEM property hierarchy again captures how much execution time is spent on different parts of the SHMEM programming model. It is either *Communication*, divided into collective and RMA, or *Synchronization* which is broken down to *Barrier*, *Point-to-Point*, *Init/Exit*, or *Memory Management*. The compound patterns *Late Broadcast*, *Wait at NxN*, *Wait at Barrier*, and *Lock Competition* are defined exactly like the corresponding MPI or OpenMP properties.

6. Conclusion and Future Work

We defined two event models describing the dynamic behavior of parallel applications involving RMA transfers. The basic model can be used for RMA implementations with blocking behavior, that is, vendor-specific one-sided communication libraries like SHMEM or language extension like Co-Array Fortran and Unified Parallel C (UPC). For MPI, we defined an extended event model that reflects the user-visible behavior as specified by the MPI-2 standard. We also defined RMA-related performance properties which represent inefficient behavior of RMA communication and synchronization. We implemented an extension to the KOJAK performance analysis toolset to instrument and trace applications based on MPI-2 and SHMEM communication and synchronization and to analyze the collected traces using the EXPERT automatic trace analysis component of KOJAK.

References

[1] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI - the Complete Reference, Volume 1, The MPI Core*. 2nd ed., MIT Press, 1998.

[2] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI - the Complete Reference, Volume 2, The MPI Extensions*. MIT Press, 1998.

[3] A. Mirin and W. Sawyer. A scalable implementation of a finite volume dynamical core in the Community Atmosphere Model. *International Journal of High Performance Computing Applications*, Vol. 19, No. 3, 203-212, 2005.

[4] K. Mohror and K.L. Karavanic. Performance Tool Support for MPI-2 on Linux. In *Proceedings of SC'04*, Pittsburgh, PA, Nov. 2004.

[5] S. Shende, A. D. Malony, J. Cuny, K. Lindlan, P. Beckman, and S. Karmesin. Portable Profiling and Tracing for Parallel Scientific Applications using C++. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, pp. 134–145. ACM, Aug. 1998.

[6] Pallas/Intel. *The Intel Trace Collector*. 2004.
→ http://www.intel.com/software/products/cluster/tcollector/

[7] F. Wolf. *Automatic Performance Analysis on Parallel Computers with SMP Nodes*. Dissertation, NIC Series, Vol. 17, Forschungszentrum Jülich, 2002.

[8] R. W. Numrich and J. K. Reid. Co-Array Fortran for Parallel Programming. *ACM Fortran Forum*, 17(2), 1998.

[9] F. Wolf and B. Mohr. Automatic Performance Analysis of Hybrid MPI/OpenMP Applications. *Journal of Systems Architecture, Special Issue 'Evolutions in parallel distributed and network-based processing'*, 49(10–11):421–439, Nov. 2003.

[10] B. Mohr, L. DeRose, and J. Vetter. A Performance Measurement Infrastructure for Co-Array Fortran. In *Proceedings of Euro-Par 2005*, Springer, LNCS 3648, pp. 146–156, Lisboa, Portugal, Sep. 2005.

[11] M.-A. Hermanns, B. Mohr, and F. Wolf. Event-Based Measurement and Analysis of One-Sided Communication. In *Proceedings of Euro-Par 2005*, Springer, LNCS 3648, pp. 156–166, Lisboa, Portugal, Sep. 2005.

[12] F. Wolf, B. Mohr, J. Dongarra, and S. Moore. Efficient Pattern Search in Large Traces through Successive Refinement. In *Proceedings of Euro-Par 2004*, Springer, LNCS 3149, pp. 47–54, Pisa, Italy, Sep. 2004.

[13] A. Kühnal. *Performance Properties for One-Sided Communication Mechanisms* (In German). Diploma Thesis. Forschungszentrum Jülich, 2005.

[14] B. Wylie, B. Mohr, F. Wolf. Holistic hardware counter performance analysis of parallel programs. In *Proceedings of Parallel Computing 2005* (ParCo 2005), Malaga, Spain, Sep 2005 .

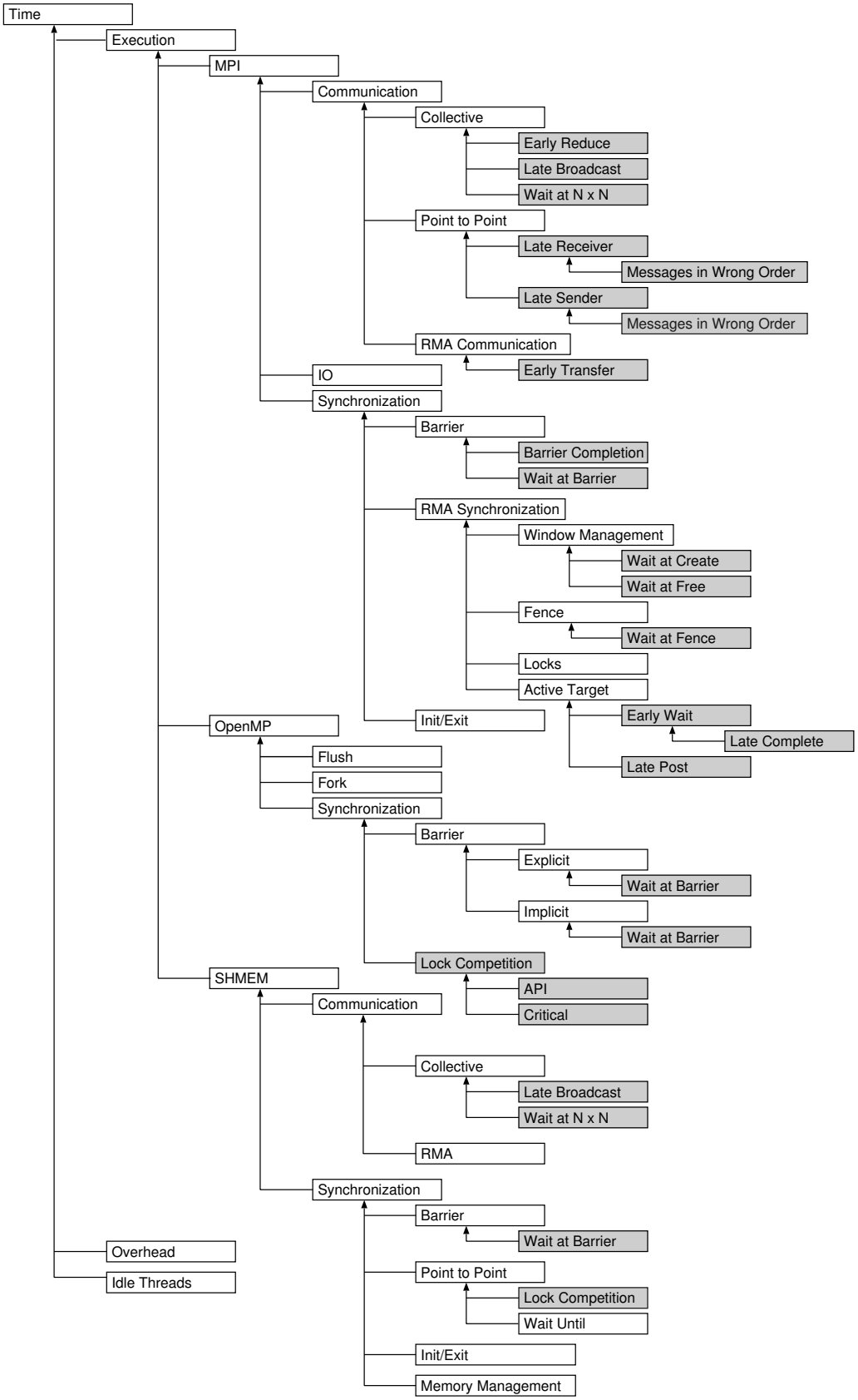


Figure 2. Performance properties defined by KOJAK

Runtime Checking of MPI Applications with MARMOT

Bettina Krammer[a], Matthias S. Müller[b] and Michael M. Resch[a]

[a]High Performance Computing Center Stuttgart (HLRS)
Allmandring 30, D-70550 Stuttgart, Germany
{krammer, resch}@hlrs.de

[b]Centre for Information Services and High Performance Computing (ZIH)
D-01062 Dresden, Germany
matthias.mueller@tu-dresden.de

The Message Passing Interface (MPI) is widely used to write parallel programs using message passing, but it does not guarantee portability between different MPI implementations. When an application runs without any problems on one platform but crashes or gives wrong results on another platform, developers tend to blame the compiler/architecture/MPI implementation. In many cases the problem is a subtle programming error in the application undetected on the platforms used previously. Finding this bug can be a very strenuous and difficult task. This paper presents MARMOT, an automated tool designed to check the correctness of MPI applications during runtime. Examples of such violations are the introduction of irreproducibility, deadlocks, incorrect management of resources such as communicators, groups, datatypes etc. or the use of non-portable constructs.

1. How to Find Bugs in Parallel Programms

Parallel programs can not only be inflicted with all the bugs known from serial programming (multiplied by the number of processes) but also by bugs resulting from the interaction of several parallel processes. Reproducibility is often not given. Finding these bugs in a complex parallel application is quite a painful task. Fortunately there are powerful tools for the different aspects of debugging, e.g. tools for memory checking or for correctness checking. Apart from the classical way of debugging – `printf` statements – the different solutions are roughly grouped into four different approaches: classical debuggers, special MPI libraries and other tools that may perform a runtime or post-mortem analysis.

1. The freely available debugger gdb [16], which is also used with its graphical front-end ddd [17], has currently no support for MPI, but it can be attached to one or several, possibly already running MPI processes. The same can be done with special memory-checking debuggers like valgrind [18,19]. More convenient are parallel debuggers, which are based on serial debuggers like gdb. They provide the usual interactive functionality of debuggers, such as single-stepping, breakpointing, evaluating variables, etc., but additionally allow the user to monitor and act on groups of processes in a single debugging session. Examples are the well-known commercial debuggers Totalview [14] or DDT [13]. These debuggers can also be used for a post-mortem analysis of core files.

2. The second approach is to provide a special debug version of the MPI library (e.g. mpich or NEC-MPI). This version is not only used to catch internal errors in the MPI library, but also to detect some incorrect usage of MPI by the user, e.g. a type mismatch of sending and receiving messages or mismatched collective operations [4–6].

3. Another possibility is to develop tools dedicated to finding problems within MPI applications at runtime. At present, three different message-checking tools are under more or less active development: MPI-CHECK [8], Umpire [3] and MARMOT [9,10]. MPI-CHECK is currently restricted to Fortran code and performs argument type checking or finds problems like deadlocks [8]. Like MARMOT, Umpire [3] uses the profiling interface.

4. The fourth approach is to perform a post-mortem analysis by collecting all information on MPI calls in a trace file. After program execution, this trace file is analysed by a separate tool or compared with the results from previous runs [7]. An example of this is the Intel Message Checker (IMC) [20]. This approach is also used by many tools with respect to performance analysis, and indeed, in some cases it can be very enlightening to "abuse" a performance tool for debugging.

As no tool is an all-in-one device suitable for every purpose, a combination of different tools will probably aid the developers most. While a memory-checking debugger may be able to diagnose that an application crashes due to an uninitialized variable, it will definitely not help you much in finding incorrect usage of the MPI interface as MARMOT does. Regardless, not every error can be caught by tools.

2. Architecture of MARMOT

MARMOT uses the so-called profiling interface to intercept MPI calls and analyse them during runtime. It does not require any modification of the application's source code nor of the MPI library, it is just a library that has to be linked to the application in addition to the underlying MPI implementation.

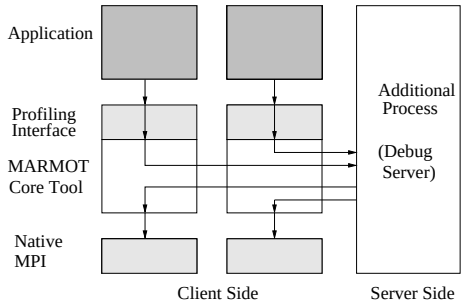


Figure 1. Design of MARMOT.

Figure 1 illustrates the design of MARMOT. For all tasks that require a global view, e.g. deadlock detection or the control of the execution flow, MARMOT uses an additional process, the so-called Debug Server. Each client registers at the debug server, which in turn gives its clients the permission for execution in a roundrobin way. In order to ensure that this additional debug process is transparent to the application, we map `MPI_COMM_WORLD` to a MARMOT communicator containing only the application processes. Since all other communicators are derived from `MPI_COMM_WORLD` they automatically exclude the debug server process. Everything that can be checked locally, e.g. verification of arguments such as tags, communicators or ranks, is performed by the client. Additionally,

the clients and the Debug Server use MPI internally to transfer information. This server/client architecture inflicts a bottleneck, thus affecting the scalability and performance of the tool, especially for communication-intensive applications [12].

3. Runtime Checking of MPI Applications

3.1. Runtime versus Post-mortem Analysis

Compared to human intervention, automatic checking has the potential to scale better. Two different scalability issues are important. First, scalability to many processors, and second, scalability in runtime. We see programs that show illegal or non-portable use of MPI only after an exceptionally long runtime. For instance, an application increases the send/receive tag after each iteration step by 1. After thousands of iterations, the tag finally exceeds the limit of up to 32767 that is guaranteed by the MPI standard. While many MPI implementations grant a much higher range of tags, there are implementations that are strict about that limit, for example versions of LAM-MPI prior to 7.0.

Since runtime checking does not require storing a huge amount of persistent information there is no scalability limit regarding runtime. Another advantage of runtime checking, compared to offline analysis, is that the application is still up and running when an error is detected. Thus other tools can be used to further analyse the situation. For example, a debugger can be attached to check the content of variables and understand the status of the application.

3.2. Frequent Problems of Parallel Programming

Parallel programming is a complex challenge. It offers enough pitfalls that MPI can imaginably stand for “Maddening Programming Interface”. Among the Top Ten common programming errors are:

- **Deadlocks:** MARMOT contains a mechanism to automatically detect deadlocks and notify the user where and why they have occurred. In general, deadlocks are caused by the non-occurrence of something else, for example mismatched send/receive operations or mismatched collective calls. One can distinguish between *real* deadlocks, which occur inevitably, and *potential* deadlocks, which may occur only under certain circumstances, e.g. depending on data races or on the implementation, for instance, if a standard send is implemented as a buffered send or not. In this code snippet process 0 and process 1 exchange messages between each other.

```
if (rank == 0) {
  // send to 1 and receive from 1
  MPI_Send(...);
  MPI_Recv(...);
} else if (rank == 1) {
  // send to 0 and receive from 0
  MPI_Send(...);
  MPI_Recv(...);
}
```

  If the `MPI_Send` is implemented in buffered mode, for example for small message sizes, this code will not deadlock, otherwise it will. Currently MARMOT’s deadlock detection is based on a timeout mechanism and therefore finds all real deadlocks. MARMOT’s debug server

surveys the time each process is waiting in an MPI call. If this time exceeds a certain user-defined limit on all processes at the same time, the debug process issues a deadlock warning. The user is then able to trace the last few calls on each node. It is also possible that attaching MARMOT (or any other tool) to an application slightly changes the execution flow in such a way that a potential deadlock becomes apparent.

- **Data races:** Potential race conditions can be caused by various reasons, e.g. by the use of a receive call with the wildcard `MPI_ANY_SOURCE` as source argument or the wildcard `MPI_ANY_TAG` as tag argument, by the use of random numbers, or by the fact that nodes do not behave exactly the same. Some users also rely on collective calls being synchronising, however, the only synchronising collective call is the `MPI_Barrier`. Other collective calls can be synchronising or not, depending on their implementation. For example, assume that any of the send calls on the processes 1 and 2 match to any of the receive calls on process 0.

```
if (rank == 0) {
  MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG,...);
  MPI_Bcast(....);
  MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG,...);
} else if (rank == 1) {
  MPI_Send(...);
  MPI_Bcast(...);
} else if (rank == 2){
  MPI_Bcast(...);
  MPI_Send(...);
}
```

  If the `MPI_Bcast` is synchronising process 0 will have to receive the message from process 1 first. If it is not then the message order will not be deterministic: either the message from process 1 or from process 2 can be received first. At present, MARMOT indicates the use of wildcards, but it does not construct dependency graphs to view the different possible executions nor does it use methods like record and replay to identify and track down bugs in parallel programs [7] or to compare different runs. Why does one need a tool to detect this sort of argument as a simple *grep* command on the source code would give the same result? Actually, a search command does neither show the execution flow nor will it be able to detect this argument if the application takes functions from some other library with hidden MPI calls.

- **Mismatches:** Mismatches in arguments of one call can be detected locally and are sometimes even detected by the compiler. Examples are wrong type or number of arguments. Mismatches are also seen in arguments involving more than one call, e.g. in send/receive pairs or in collective calls. Special attention is needed when comparing matched pairs of derived datatypes because it is legal to send, for example two (`MPI_INT`, `MPI_DOUBLE`) and to receive one (`MPI_INT`, `MPI_DOUBLE`, `MPI_INT`, `MPI_DOUBLE`), or to send one (`MPI_INT`, `MPI_DOUBLE`) and to receive one (`MPI_INT`, `MPI_DOUBLE`, `MPI_INT`, `MPI_DOUBLE`) (a so-called *partial* receive). MPI implementations usually abort an application when there is a datatype mismatch, e.g. send an `MPI_INT` and receive an `MPI_DOUBLE`, but no exact diagnosis of the mismatch is given.

- **Resource handling:** This is an area in MPI where incorrect usage may result in fatal errors with almost no obvious link to the real cause. Since they are very difficult to find, we place special focus into detecting them. MARMOT is able to keep track of the proper construction, usage and destruction of all MPI resources, such as communicators, groups, datatypes, etc. As these resources are "opaque" objects and therefore implementation-dependent, MARMOT has its own book-keeping of these resources and, thus, duplicates the management done by the underlying MPI library. MARMOT also checks if requests and other arguments (tags, ranks, etc.) are used correctly, e.g. if an active requests is reused. The main functionality is implemented for the C language binding, whereas the functionality for the Fortran language binding is obtained through a wrapper to the C interface. Special attention is paid to the verification of the datatypes because they are one of the major differences between the C and the Fortran language binding.

- **Memory and other resource exhaustion:** Non-blocking calls such as `MPI_Isend` etc. can complete without issuing a matching test or wait call. However, the number of available request handles is limited (and implementation defined). Therefore requests should always be freed, as should allocated communicators, datatypes, etc. MARMOT gives a warning when a request is reused, and also when there are active or non-freed requests left at the `MPI_Finalize`. Another issue is reusing memory that is still in use, for example by reading/writing from/into a buffer by an unfinished send/receive operation. MARMOT does currently not perform any checks if a buffer can be reused safely because the transmission of data has completed. This kind of check is a subtle task that requires some insight into an MPI implementation: what is really going on when calling e.g. `MPI_Issend` or `MPI_Irecv`, how does that depend on the message size, etc.? In some cases, MARMOT checks if buffers are overwritten by mistake, e.g. for `MPI_Gatherv` and similar collective calls, it is verified if on the root process data is overridden due to an erroneous array of displacements.

- **Portability:** The MPI standard leaves many decisions to the implementors, for example how to implement opaque objects and handles to these objects, if to implement `MPI_Send` as buffered call or not, if to implement collective calls as synchronising calls, if to make the implementation thread-safe or not, etc. Some of these issues can already be detected at compile time when the application is ported to another environment, some can be found at runtime by MARMOT, e.g. using a tag beyond the guaranteed limit. Another example of non-portable constructs can even be found in the MPI-1 standard on pages 79 - 80 (and we have found it exactly like that in a user's code):

```
 /* build datatype describing structure */
 ...
 MPI_Aint disp[3];
 int      base;

/* compute displacements of structure components */
...
base = disp[0];
for (i=0; i<3; i++) disp[i] -= base;

MPI_Type_struct(...,disp,...);
```

If you ever try to use the datatype constructed above on a 64-bit-architecture the code will probably crash without any warning, not at the construction step, but later on when the datatype is used, e.g. in a send/receive call. The reason is that `MPI_Aint` is `long` and not `int` there, and thus, some significant bits are lost in the computed array of displacements.

MARMOT supports the complete MPI-1.2 standard, although not all possible tests (such as consistency checks) are implemented. It can be used with any standard-conforming MPI implementation and may thus be deployed on any development platform available to the programmer. Although high-quality MPI implementations detect some of these errors themselves, there are many cases where they do not give any warnings. For example, non-portable implementation-specific behaviour is not indicated by the implementation itself, nor are checks performed that would decrease the performance too much, such as consistency checks. What is worse, MPI implementations tolerate quite a few errors without warnings or crashing, by simply giving wrong results.

MARMOT is tested on Linux Clusters with IA32/IA64 processors, Cray, Hitachi, IBM Regatta and NEC SX systems, using different compilers (GNU, Intel, PGI,etc.) and different MPI implementations (mpich, LAM/MPI, vendor MPIs, etc.). Functionality and performance tests are performed with test suites, microbenchmarks and real applications [11,12].

4. More Examples

There are many examples of errors that are tolerated by MPI implementations or that only occur on specific platforms, or occur under specific circumstances. For example, in mpich it is possible to use the Fortran datatype `MPI_INTEGER` in a C program without any problems or warnings because, in this implementation, some Fortran datatypes are defined in the C include file `mpi.h`. Implementations that are stricter, e.g. LAM/MPI, will not execute that code. When analysing the development version of a medical application that uses a 3D Lattice-Boltzmann method for blood flow calculation, we find another portability problem of that kind. In many places the developers equate `MPI_Comm` with `int`. This is a dangerous thing to do because, in mpich, the opaque object `MPI_Comm` is actually defined as an `int` and therefore the code works without a problem. However, in LAM/MPI, `MPI_Comm` is defined as a pointer to a `struct` and therefore it breaks on any platform where a pointer does not fit into an integer.

When we test this application with different input files representing the geometry of the artery we find other problems. In the simplest case, a mere tube with an approximately constant radius, the code runs without any problems. When calculating the blood flow for an artery stenosis, i.e. using a tube with varying radius, the application stalls and MARMOT finds a deadlock caused by process 0, which performs an `MPI_Sendrecv` whereas all other processes perform an `MPI_Bcast`. A very simplified skeleton of the source code shows why:

```
main {
  ...
  // compute number of iterations depending on the radius
  if (radius < x) num_iter = y; else num_iter = z;

  for (i=0; i < num_iter; i++)
  {
    // compute blood flow and exchange results
    // with neighbours using MPI_Sendrecv
```

```
      computeBloodflow(...);
    }

    // communicate results using MPI_Bcast
    writeResults(...);
  }
```

Every process calculates its own number of iterations depending on the radius, but unfortunately, they do not communicate to agree on a maximum number of iterations. As a result, process 0 tries to perform more iterations having a piece of the artery with a bigger radius than the others, and therefore tries to exchange results with its neighbour using the `MPI_Sendrecv` whereas the others already have finished their iterations and try to communicate with the `MPI_Bcast`. This shows how important it is to choose relevant input data sets for an effective runtime checking. It also shows how difficult it is for developers to keep track of all the MPI communication when it is hidden in subroutines.

Another input file representing a forked artery reveals yet another programming error. In this case, every process results in different values for the send/receive counts in the collective call `MPI_Gather`. On some platforms the application runs without a problem, but on some platforms the different values cause a segmentation violation. Strangely enough, on one platform the application runs without a problem, but when attaching a performance analysis tool to it, it crashes. It appears to be the fault of the tool, but in reality there is a bug in the application. Antithetically, it is possible that bugs never occur in the presence of tools (so-called *Heisenbugs*).

5. Conclusions and Future Work

Hitherto, we have presented the MARMOT tool, which analyses the behaviour of an MPI application during runtime and checks for errors frequently made in the use of the MPI API. The functionality of this tool has been tested with real world applications.

Although there is still plenty of work to do, we believe that MARMOT is on the right track and will become an indispensable tool in the development of MPI-parallel applications. Future work includes an extension of MARMOT's functionality according to the users' needs. To offer a more user-friendly interface, to support frequently used parts of MPI-2 such as parallel file I/O [10] or to support hybrid applications written in OpenMP and MPI. Another goal is to improve the performance and scalability of the tool, especially for communication-intensive applications.

Acknowledgments

The development of MARMOT was partially supported by the European Union through the IST-2001-32243 project "CrossGrid".

References

[1] Message Passing Interface Forum. *MPI: A Message Passing Interface Standard*, June 1995. `http://www.mpi-forum.org/`.

[2] Message Passing Interface Forum. *MPI-2: Extensions to the Message Passing Interface*, July 1997. `http://www.mpi-forum.org/`.

[3] J.S. Vetter and B.R. de Supinski. Dynamic Software Testing of MPI Applications with Umpire. In *Proceedings of the 2000 ACM/IEEE Supercomputing Conference (SC 2000)*, Dallas, Texas, 2000.

[4] William D. Gropp. Runtime Checking Of Datatype Signatures In MPI. In *Recent Advances In Parallel Virtual Machine And Message Passing. 7th European PVM/MPI Users' Group Meeting. LNCS 1908*, pages 160-167. Springer 2000.

[5] Chris Falzone, Anthony Chan, Ewing Lusk and William Gropp. Collective Error Detection for MPI Collective Operations. In *Recent Advances In Parallel Virtual Machine And Message Passing. 12th European PVM/MPI Users' Group Meeting. LNCS 3666*, pages 138-147. Springer 2005.

[6] J.L. Träff and J. Worringen. Verifying Collective MPI Calls. In *Recent Advances In Parallel Virtual Machine And Message Passing. 11th European PVM/MPI Users' Group Meeting. LNCS 3241*, pages 18 - 27, Springer, 2004.

[7] Dieter Kranzlmüller. *Event Graph Analysis For Debugging Massively Parallel Programs.* Phd thesis, Joh. Kepler University Linz, Austria, 2000.

[8] Glenn Luecke, Yan Zou, James Coyle, Jim Hoekstra and Marina Kraeva. Deadlock Detection In MPI Programs. In *Concurrency and Computation: Practice and Experience.* 2002, vol. 14, pages 911 - 932.

[9] MARMOT. `http://www.hlrs.de/organization/tsc/projects/marmot`

[10] Bettina Krammer, Matthias S. Müller and Michael M. Resch. MPI I/O Analysis and Error Detection with MARMOT. In *Recent Advances In Parallel Virtual Machine And Message Passing. 11th European PVM/MPI Users' Group Meeting. LNCS 3241*, pages 242 - 250, Springer, 2004.

[11] Bettina Krammer, Katrin Bidmon, Matthias S. Müller, and Michael M. Resch. MARMOT: An MPI analysis and checking tool. In *Proceedings of PARCO 2003*, pages 493-500, Elsevier, 2004.

[12] Bettina Krammer, Matthias S. Müller and Michael M. Resch. MPI Application Development Using the Analysis Tool MARMOT, In *Proceedings of ICCS 2004, LNCS 3038*, pages 464 - 471, Springer 2004.

[13] DDT. The Distributed Debugging Tool.
`http://www.streamline-computing.com/softwaredivision_1.shtml`

[14] Totalview. `http://www.etnus.com/Products/TotalView`

[15] mpigdb.
`http://www-unix.mcs.anl.gov/mpi/mpich/docs/userguide/node26.htm#Node29`

[16] The GNU Project Debugger. `http://www.gnu.org/manual/gdb`

[17] The Data Display Debugger. `http://www.gnu.org/software/ddd/`

[18] Brett Carson and Ian A. Mason. ClusterGrind: Valgrinding LAM/MPI Applications. In *Recent Advances In Parallel Virtual Machine And Message Passing. 12th European PVM/MPI Users' Group Meeting. LNCS 3666*, pages 325-332. Springer 2005.

[19] Valgrind support for MPIch.
`http://www.hlrs.de/people/keller/MPI/mpich_valgrind.html`

[20] Jayant DeSouza, Bob Kuhn and Bronis R. de Supinski. Automated, scalable debugging of MPI programs with Intel Message Checker. *SE-HPCS '05*, St. Louis, Missouri, USA. `http://csdl.ics.hawaii.edu/se-hpcs/papers/11.pdf`

[21] A. Tirado-Ramos, H. Ragas, D. Shamonin, H. Rosmanith, and D. Kranzlmueller. Integration of blood flow visualization on the grid: the flowfish/gvk approach. In *2nd European Across Grids Conference*, Nicosia, Cyprus, January 28-30 2004.

Automated Correctness Analysis of MPI Programs with Intel®Message Checker

Victor Samofalov[a], Victor Krukov[b], Bob Kuhn[c], Sergey Zheltov[c], Alexandr Konovalov[c], Jayant DeSouza[c]

[a]Intel Corporation

[b]Russian Academy of Sciences

[c]Intel Corporation

1. Introduction

Parallel programming is widely considered to be much more complex than sequential programming. When implementing multi-threaded or multi-process parallel algorithms new kinds of errors related to the simultaneous use of shared resources emerge. In addition to introducing new kinds of synchronization errors, the non-determinism of parallel programs can lead to errors that occur intermittently and are hard to reproduce on-demand. Furthermore, old ”sequential” errors can be erroneously interpreted as ”parallel” ones, e.g. writing the wrong value to a global variable is harder to track down in a multi-threaded program. These and other factors significantly complicate implementation and debugging of parallel programs.

For larger parallel systems, distributed parallel programming is the most effective way to improve computing performance. Distributed parallel cluster systems currently dominate high-performance computing, constituting over 70% of the 2004 Top500 list. In most cases, distributed parallel programs use the message-passing programming paradigm, and of the various available candidates, portability and performance have led to MPI as the *de facto* standard for cluster programming. However, the variety and complexity of MPI operations (about 200 in the MPI 1.2 standard) has led to the introduction of new kinds of program errors.

This paper examines the specific area of correctness analysis for MPI programs and describes a new correctness tool developed at Intel’s Advanced Computing Center, called Intel® Message Checker (IMC).[1] Intel® Message Checker is a unique tool for the *automated* analysis of MPI programs. It analyzes trace files and detects several kinds of errors with point-to-point and collective operations such as (a) mismatches in message/buffer sizes, data types, and MPI resources, (b) race conditions, and (c) deadlocks and system-buffer related deadlocks. In addition to the comprehensive analysis engine, IMC also features a graphical user interface with broad functionality for program and data representation. IMC can significantly assist an MPI programmer with distributed program analysis and debugging, and helps them find issues earlier in the debugging process and in less time.

An advantage of *trace-based analysis* is that, if it is sufficiently fast and non-intrusive, it can be used to catch intermittent errors by enabling it for all production runs. Message Checker has low perturbation, which allows it to be used on production runs and the trace-based approach is well-suited to finding subtle problems which surface intermittently on runs.

Fortunately, because the nature of message passing reduces shared data, indeterministic and irreproducible errors are not so common for distributed MPI programs as for shared memory ones. But there are other issues unique to large distributed programs — scalability issues for large systems (how does a user find a problem in 500 processes?) and the need for constant performance optimization. A clear advantage of *automated analysis* is that it scales to extremely large systems better than

current-generation debugging tools (the almighty `printf`).

We note also that automated correctness checking tools (such as IMC) can not only be used for debugging, i.e. assisting the programmer to find an error, but go one step further and actually find the error. As such, this new class of tools, which we refer to as *confidence tools*, can help ensure that a program that provides correct results is really correct. Message Checker is widely used by Intel engineers and has been used to detect issues in several non-trivial MPI applications.

In addition to indicating the error, an advanced correctness checking tool should provide maximum information about the program run to aid the programmer in detecting the cause of the problem. Information only about the stack frame of a failed MPI call, which is what is usually available in debuggers, is not enough for effective bug fixing because the user is unable to answer a non-trivial but very important question — "How did we reach this state in the program?". A call stack provides the current call path, but not the history of calls that have led to the program reaching this state, i.e. it provides depth, but not breadth. Well-known interactive debugging tools for large parallel systems do not make the situation clearer for users. But the *call history* is easily provided by a trace-based post-mortem tool such as IMC. The ideal would be a blend of interactive and trace-based program analysis for MPI correctness.

The rest of this paper is organized as follows. MPI's communication primitives provide opportunities to check correctness on various levels as described in the next section. Our initial performance results are described in Section 4.

2. Taxonomy of Detected Errors by Locality of Analysis

By definition, a programming error is a difference between actual behavior and desired behavior (i.e., specification). From the analysis point of view, errors detected by Intel® Message Checker can be divided into four categories:

1. Errors detection which requires only local MPI function information. e.g. a derived data type with overlapping elements on the receiving side, since such types are valid only for send operations.

2. Errors that can be detected by analyzing only local process information, e.g. initiating a non-blocking point-to-point operation without waiting for or freeing the request.

3. Errors related to multiple processes inside one communicator. There are two sub-classes here:

   (a) Point-to-point errors e.g. incompatibility between data types in a send and its corresponding receive

   (b) Collective operation errors, e.g. different processes specifying different reduce operation in an `MPI_Reduce()`.

4. Error detection which does not require information about inter-process communication but requires information about the states of processes instead. The most serious errors in this category are so-called *potential deadlocks*. In our terminology, a potential deadlock is a deadlock that may become actual if the internal behavior of a function changed, e.g. an `MPI_Send()` for 4-byte message can be either blocking or non-blocking depending on the choice of the MPI implementation.

The current version of Intel® Message Checker supports the semantics of the MPI-1.2 standard. In future, IMC will be extended to support MPI-2.

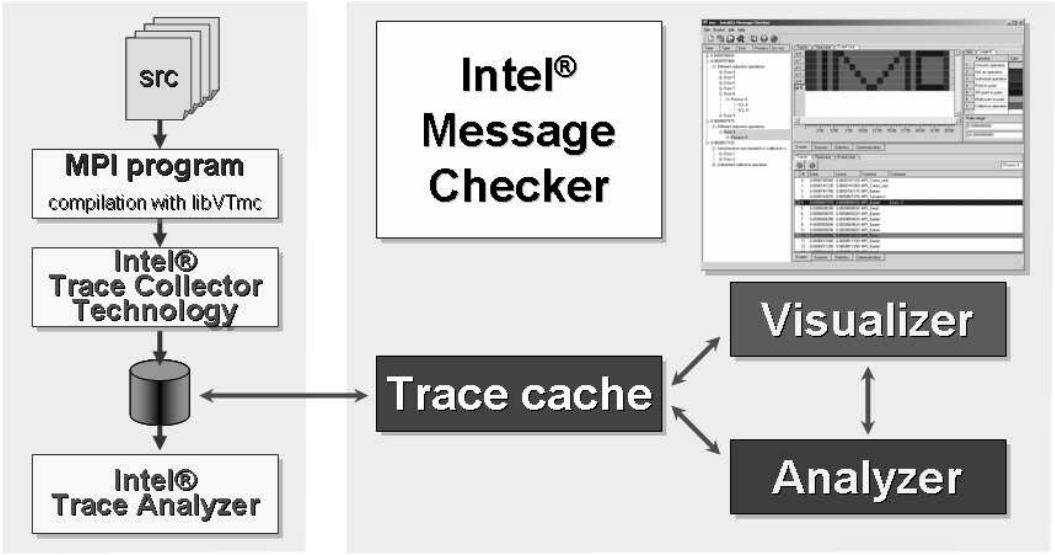


Figure 1. General Intel® Message Checker Software Structure.

As an aside, we note that some error checking can be performed inside the MPI library. An interesting example of such an approach is re-mapping communicators, datatypes, etc. in MPICH2 into integer numbers that are then used for error-checking instead of the original types. This way the MPI library can easily detect invalid arguments (communicators, datatypes, etc.) during an MPI function call.

3. Architecture of Intel® Message Checker and Other Approaches to Implementation

The general software architecture of IMC is presented on Fig. 1. At the linking stage, the MPI program is instrumented with a special version of the Intel® Trace Collector library (the PMPI interface is used inside). Then trace data is collected during the program run and saved into a trace file. Correctness analysis can be performed by either a command-line tool or in the GUI Visualizer environment after the program finishes. The command-line version is targeted for batch/automated testing; the GUI is designated for interactive use in a debugging session.

3.1. Analysis Approach in IMC and Comparison With Related Tools

Other MPI checking tools like Marmot [5] can detect the same set of errors as IMC except deadlocks (the timeout method employed by Marmot cannot guarantee that a real deadlock was detected, and cannot reconstruct the loop in the resource graph which is important for the user to find the real problem). The main difference between Marmot and IMC is in the approach, i.e. Marmot uses an online interactive approach, whereas IMC uses a trace-based offline/post-mortem approach.

In the upcoming MPICH2 release a collective operations checking library [3] will be available but with less functionality then IMC has for such operations.

Both suffice for the “local” error cases described above. But, more complicated error detection requires a more aggressive use of non-local contexts. There are two approaches for this: run-time validity checking or post-mortem trace processing. Most of the existing tools (Marmot [5], collchk [3], NEC run-time MPI checking library [7]) employ run-time analysis.

A significant advantage of run-time checking is the interactive and onine response, i.e. the user

can see the problem report directly as it was found, without spending hours running the program to collect a trace.

But run-time provided error diagnostics may have one disadvantage for the user — they may be not able to see the call history (see Introduction) of the distributed parallel task. Furthermore, the efficient run-time detection of ”distributed” errors and error-prone situations (like deadlocks and race conditions) is quite complex [2]. Inter-process error detection leads to additional communication overhead, which can be quite significant sometimes. These deficiencies are overcome by IMC’s trace-based and post-mortem analysis.

On the other hand, the main disadvantage of the post-mortem approach is the possibility of huge trace files; even modest parallel tasks can generate traces with hundreds of millions of events and of many gigabytes in size. So, tools for post-mortem correctness checking should efficiently support such huge data volumes.

3.2. Visualization Approaches in IMC

In program debugging it is very important for the user to understand how his program appears to be in some state. Saving the history of all program state changes (such as changes to every variable) is unacceptable and also distracts the user with excessive information. One possible solution to this problem is the automatic building of the program’s abstraction model using low-level data and reverse engineering of high-level models. A survey of existing approaches to restoring high-level states can be found in [4]. Such approaches originally were applied to the object-oriented programming model but in general can also be applied to the procedural message-passing paradigm (for example, a call to an object method can be considered as the sending of a message to this object).

Luckily message-passing programming has number of alternative solutions in this area. Visualization methods for message-passing traces have been in development for over 25 years, and are quite mature. They are mostly used in performance analysis, but such a position is too restrictive. Indeed, Gantt chart-based views have many times demonstrated their usefulness during understanding ”what’s going on in the parallel task”.

There are two main views for program structure presentation in IMC Visualizer (see Fig. 2).

1. The Time-Line View — represents actual MPI function calls and the interaction of processes over time. This view reflects the native picture of program execution and may be used to track excessive delays and errors resulting from an unexpected execution order.

2. The Event-Line View — represents the program execution flow and process interaction as a logical sequence of MPI operations. That is, all MPI operations are ordered using some defined relations between operations. For example, all collective operations serializing the execution of multiple processes (`MPI_Barrier`) have the same event number for all processes inside the communicator. This simplifies the analysis of processes’ inconsistent behavior.

4. Performance Results

In the current version of IMC, the user must perform several steps for program correctness checking. First, the user needs to compile their program with a special version of Intel® Trace Collector that is used for correctness data collection. (ITC Message Checker Library (ITCMCL) is a special version of ITC for extended trace data collection used for correctness checking but because of that it is not recommended for accurate performance analysis). Intel® Trace Collector is implemented as overloaded MPI function calls which write data to memory buffers and save them in a background thread to a local temporary file.

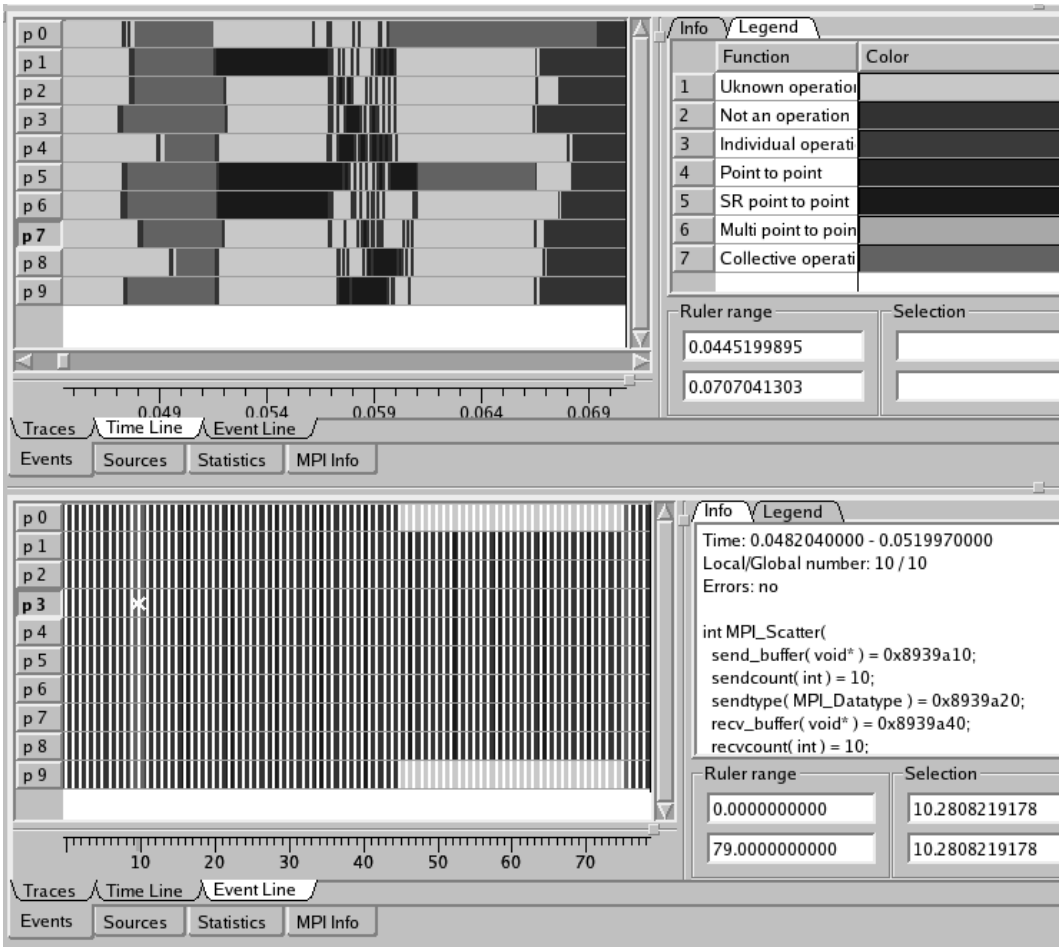


Figure 2. Time Line and Event Line Views.

In Figs. 3–4 we present data for NAS Parallel Benchmarks 2.4 in the Intel MPI Benchmarks(IMB) suite. As can be seen, the overhead of the ITCMCL library is quite small. We should note that standard MPI microbenchmarks like IMB provide a somewhat too optimistic estimation of tracing overhead because they are not aware of ITC trace collection specifics.

Our testbed was a 3-node cluster; each node had four Intel ® Xeon^{TM} 1.4 GHz CPU's with HypherThreading enabled; the interconnect used was Myrinet. The anomalous advantage of 8-process runs over 4-process runs for some benchmarks is caused by a memory bottleneck since the 4-process variant runs on a single node, and 8(9)-process ones run on two nodes).

Some observations from the real-world usage of Intel® Message Checker are:

1. IMC is most usable during program development for debugging and correctness analysis.

2. Even for stable real-word applications, running IMC on ten of them found overlapping of 1-byte receive buffers in two applications (one can be considered a benign fault because the process waited for any event from the other process and did not use the data sent.) This demonstrates the usefulness of confidence tools.

3. IMC is useful even in situation when an error was found by other tools (e.g. MPI itself) because IMC has the program history information for analysis, e.g. when IMC was used on

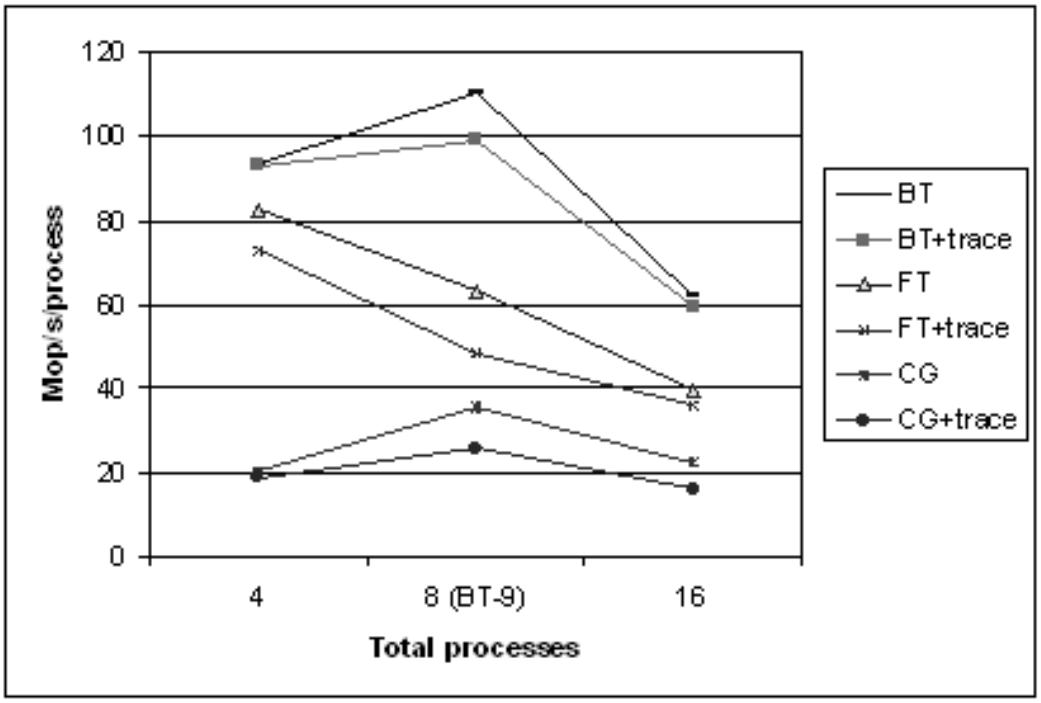


Figure 3. Overhead of correctness checking trace collection: BT, FT, and CG benchmarks.

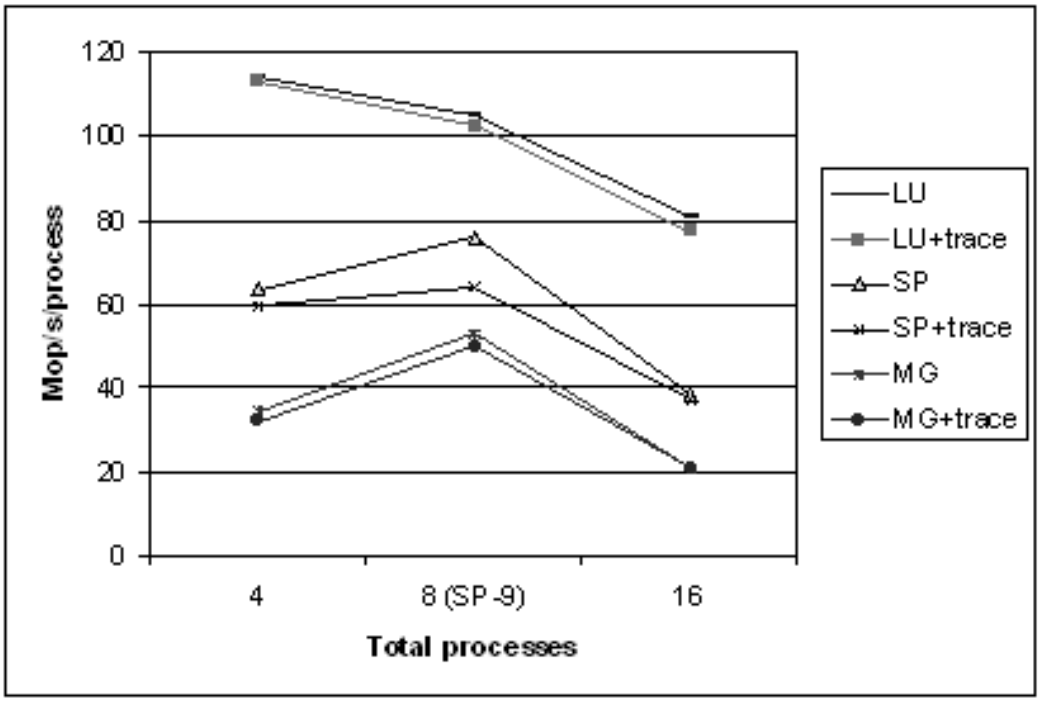


Figure 4. Overhead of correctness checking trace collection: LU, SP, and MG benchmarks.

the GAMESS [6] application — the `MPI_Allgather()` function call with different buffer lengths was first detected by Intel® MPI).

5. Conclusion

We have described the complexity of distributed parallel programming in MPI and motivated the need for a new correctness tool, the Intel® Message Checker from Intel®'s Advanced Computing Center. This tool features a trace-based approach that has low perturbation, high scalability (due to the avoidance of online global analyses), and provides a call history that complements the call stack provided by debuggers. IMC also features an automated analysis that goes futher than a debugger and actually detects errors rather than pointing out symptoms; therefore it has been useful to run it on apparently correct programs. We coined the term *confidence tools* to reflect the new role of such automated correctness tools. The trace-based approach is also useful for catching intermittent errors in large (time or CPU) runs, and the automated analysis provides advantages over manual debugging techniques, especially for large systems. IMC's GUI features an event-line view that separates out logical MPI relationships from the time-centric timeline view.

References

[1] Jayant DeSouza, Bob Kuhn, Bronis R. de Supinski, Victor Samofalov, Sergey Zheltov, Stanislav Bratanov. Automated, scalable debugging of MPI programs with Intel Message Checker. // Second International Workshop on Software Engineering for High Performance Computing System Applications, May 2005, St. Louis, Missouri.

[2] R.Cypher, E.Leu. Efficient race detection for message-passing programs with nonblocking sends and receives // Proceedings of the 7th IEEE Symposium on Parallel and Distributeed Processing, 1995.

[3] C.Falzone, A.Chan, E.Lusk, W.Gropp. Collective Error Detection for MPI Collective Operations // Euro PVMMPI'05, 2005.

[4] A.Hamou-Lhadj, T.C.Lethbridge. A survey of trace exploration tools and techniques // Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research Markham, Ontario, Canada, 2004.

[5] B.Krammer, M.S.Müller, M.M.Resch. MPI Application Development Using the Analysis Tool MARMOT // ICCS 2004, Krakow, Poland, June 7-9, 2004. Lecture Notes in Computer Science, Vol. 3038, pp. 464–471, Springer, 2004.

[6] M.W.Schmidt, K.K.Baldridge, J.A.Boatz, S.T.Elbert, M.S.Gordon, J.H.Jensen, S.Koseki, N.Matsunaga, K.A.Nguyen, S.Su, T.L.Windus, M.Dupuis, J.A.Montgomery General Atomic and Molecular Electronic Structure System // J. Comput. Chem., 14, 1347-1363(1993).

[7] Jesper Larsson Träff, Joachim Worringen. Verifying Collective MPI Calls // Recent Advances in Parallel Virtual Machine and Message Passing Interface, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, September 19-22, 2004, Proceedings. Lecture Notes in Computer Science, Vol. 3241, pp. 18–27, Springer, 2004.

908

The MPI/SX Collectives Verification Library

Jesper Larsson Träff[a], Joachim Worringen[a]

[a]C&C Research Laboratories, NEC Europe Ltd., Rathausallee 10, D-53757 Sankt Augustin, Germany

This paper summarizes and discusses the functionality of an extended MPI library for verifying correct use and consistency of all collective functions of the MPI-2 standard. The library is part of the MPI/SX implementation for the NEC SX-series of parallel vector-computers, as well as NEC implementations of MPI for other platforms. We give examples of the use of the verification library, and in particular report on the overheads entailed. The library could have been implemented as a stand-alone, portable interface by using the MPI "profiling interface" as defined by the MPI standard. We discuss obstacles of a portable implementation, and instead argue to support collective verification as part of any good MPI implementation.

1. Introduction

MPI, the *Message Passing Interface* MPI [4, 9], contains a large number of functions that are collective over a set of processes, meaning that all the processes in the set must call the function in order to let it complete on all processes. This includes collective communication and reduction operations like `MPI_Bcast` and `MPI_Reduce`, but also functions for creating new sets of processes (*communicators*), spawning processes, creating windows for one-sided communication or opening files are collective in this sense. All collective functions of MPI explicitly or implicitly pose certain *consistency requirements* among the parameters passed by the different processes, and sticking to these rules is essential for correct execution of the application. As a simple example, the processes calling the `MPI_Bcast` function must supply the same `root` parameter. Violating consistency conditions may lead to unpredictable, but often fatal behavior of the application, entirely dependent, however, on the MPI implementation at hand. The application may deadlock (as could be the case if a different `root` parameter was given in an `MPI_Bcast` or `MPI_Reduce` call), give wrong results (as could be the case if non-matching datatypes and counts were given in the call), or crash, possibly at some time later in the application (this could be the case if different process group parameters were given to a communicator creating function). On the other hand, especially for irregular collectives like `MPI_Gatherv` or `MPI_Reduce_scatter`, the consistency requirements are quite complex, and it is easy to make mistakes. Thus, it would help an application developer to have the MPI library check parameter consistency in the use of the collective MPI functions.

Consistency requirements are conditions that require communication to check, and are thus expensive compared to simple argument checks that can be performed locally by each process (is the root argument in range? is the datatype committed?). For high-performance use such checks are therefore prohibitive. Consequently, the MPI standard does not specify a behavior in case of errors, and it is legal for an MPI implementation to deadlock or even crash if collective functions are not called consistently. A solution to the dilemma would be to have a separate MPI library to be used during application development/debugging to catch consistency errors. Such a library could either be implemented stand-alone and in a portable fashion using the profiling interface feature of the MPI standard, or implemented as a separate library for a specific MPI implementation. The latter course has been followed for MPI/SX.

2. Related Work

The portable tool *MARMOT* is described in a series of papers [5, 6, 7]. It uses a single (additional) MPI process to act as a control process: the MPI processes running the user application log all MPI function calls with this control process, which in turn performs a global analysis of the function calls and the communication patterns. This allows to give detailed diagnostic output i.e. for a deadlock situation. However, this approach is inherently non-scalable, and we believe that it is possible to achieve a similar verification quality following our distributed approach if a separate deadlock-detection mechanism is available.

Umpire [12] follows a similar approach as MARMOT, but relies on out-of-band shared-memory communication to have a dedicated thread of process 0 analyze the MPI function calls of all other processes that they logged in the shared memory area. The main focus of Umpire is deadlock detection. Obviously, the requirement that all processes of the application to be verified run on a single node limits the scalability and applicability of this approach especially on cluster machines with few CPUs per node.

A deadlock and use checker for MPI programs in Fortran 90 and Fortran 77 named *MPI-CHECK* is described in [8]. Next to some checks of correct use of MPI functions which are done in part by instrumenting the Fortran code at source level, its main focus is on distributed deadlock detection. It does not use a central control instance, but instead uses a handshake-protocol on top of the MPI point-to-point communication functions to validate each send and receive call with its destination and source, respectively.

These tools do deadlock checking, which for MPI/SX is left to the MPI implementation (suspend/resume mechanism). Extensive local checks are also performed by the MPI/SX library. Thus the MPI/SX verification library focuses entirely on collective consistency requirements.

Intel Message Checker [1] does mostly checking for point-to-point communication and is an offline, trace-based tool. This means that the verification is necessarily a three-step process: first run the application to create the trace file, then run the analyzer software with this trace file as input, and finally visualize the results using a graphical tool. This indirect approach complicates the verification for the user, and depending on the size of the generated trace, may even make it infeasible.

Recently MPICH2 has incorporated a checking interface similar to the approach of MPI/SX. The MPICH2 approach is portable by using the MPI profiling interface [3]. It extends our approach by doing datatype signature checking using hash-values. On the other hand, it does not perform a verification as complete as our approach as it is not able to decode opaque MPI objects like `MPI_Group` or `MPI_Win`, does not handle inter-communicators, and is missing some other verifications, i.e. in MPI-IO (see Chapter 5).

We summarize the comparison of the different approaches in Table 1, which is based on published information. It shows the general architecture of an approach, lists the supported types of communicators for collective operations, the type of checks performed for point-to-point operations, the availability of the profiling interface when using the verification library, the portability of the software to different MPI libraries, the occasions on which the use of MPI datatypes is verified, the capability of checking the use of opaque MPI objects (like `MPI_Win` and `MPI_Group`) and the degree of support for the MPI-2 standard.

3. Design and Implementation of the MPI/SX verification library

The approach to verification in MPI/SX is described in more detail in [11]. Local checks that can be performed fast (in constant time, independent of the number of processes, data size etc.) are

Table 1
Key characteristics of different MPI verification approaches.

	architecture	collective	point-to-point checking	PMPI
NEC MPI/SX	distributed	intra & inter	deadlock	available
MPICH2	distributed	intra	datatype	used
MARMOT	centralized (distributed memory)	intra	deadlock	used
Umpire	centralized (shared memory)	intra	deadlock, buffer	used
MPI-CHECK	distributed, instrumentation	intra	deadlock	used
Intel MC	tracefile (offline)	intra	deadlock, buffer,datatype	used
	portable	opaque objects	datatype checking	MPI-2
NEC MPI/SX	no (NEC MPI)	yes	setup of file view	full
MPICH2	yes	no	communication (hash)	partially
MARMOT	yes	yes	construction	no
Umpire	limited (SMP only)	no	no	no
MPI-CHECK	limited (Fortran only)	no	no	no
Intel MC	limited (Intel MPI & MPICH2)	unknown	communication (partly)	partially

always performed by MPI/SX, as they are often valuable for catching irritating errors (e.g. rank out of range, non-committed datatype) and do not hamper performance. Non-local consistency checks on arguments to collective operations all require (expensive) communication, and are therefore exclusively done by the verification library. To verify that all processes calling e.g. `MPI_Bcast` with the same `root` argument it suffices for some process, say rank 0, to broadcast its value of the `root` argument to the other processes, which in turn check this against their own value for the `root` argument. A process detecting an inconsistency could report the error, in which case it would normally make sense to abort the application. In the spirit of MPI the error handler associated with the communicator of the `MPI_Bcast` call should be called. In case the user has changed the error handler to not abort, for instance by using `MPI_ERRORS_RETURN` this could again lead to highly unpredictable behavior, since only the processes that detected the wrong `root` argument would be aware of the error condition. To avoid this, the action of the MPI/SX verification library is implemented to be *symmetric*. All processes will be informed of a possible error condition and can thus all invoke the error handler. The cost of this an extra `MPI_Allreduce` for each verified condition. Overall, the total cost per collective operation is from two to eight extra collective calls (either `MPI_Bcast`, `MPI_Gather`, `MPI_Alltoall`, or `MPI_Allreduce`), all with small data (from a single `MPI_Aint` up to as many `MPI_Aint` as there are processes in the communicator).

Collective verification of this sort can in principle be implemented in MPI itself, and made available in a portable fashion by using the profiling interface mechanism of MPI. However there are some tedious obstacles to this approach. A minor problem is that some MPI objects (for instance `MPI_Aint`, `MPI_Op`) are not first-class citizens, and therefore (formally) cannot be exchanged in communication operations. These must therefore be mapped to objects that can be used in communication operations. More severe difficulties of this sort are the extraction of the processes from an `MPI_Group` object which is necessary when verifying for instance `MPI_Comm_create`, or the extraction of the underlying communicator from a one-sided communication window. These difficulties are trivial to overcome from within an actual MPI implementation. A further advantage of having a special library is that the profiling interface is not “used up”, meaning that other profiling can be done together with the verification library.

4. Using the verification library

To use the library, only relinking of the aplication with `-lvmpi` is needed. It is also possible to use a version of the verification library that provides a profiling interface by using `-lvpmpich`. In this case, the additional collective operations performed by the verification library will *not* be visible to the library using the profiling interface. This is due to the fact that the verification library uses internal hooks to these operations.

The level of verification and/or reporting is controlled either by the MPI profiling interface function `MPI_Pcontrol(level,...)` or by setting the environment variable `MPIVERIFY`.

- `level 0`: disabled
- `level 1`: return an error code to the error handler
- `level 2`: print an additional description of the problem

By default, the verification level is set to 2. Setting the level to 0 will cause only minimal run-time overhead of a few `if` statements per collective operation. Depending on the individual requirements, this allows to use the verification library also for production code. All other verification levels will, if a problem occurs, call the active error handler of the communicator in whose context the function was called. In such a case the error handler is called by all processes.

5. Examples

We have tested the verification library with some existing applications. Finding errors in an application in production is expected to be rare, as these have already been tested by other means.

We expect the verification library to be more successful when it is used during application development. We could resolve a bug in such an application where a series of `MPI_Bcast` operations was performed. The first operation broadcasted the amount of data for the following operations. However, there was a mismatch in these values on the root process by which too little data was broadcasted in the subsequent operations. This problem was not discovered within the broadcast operations, but showed up much later in the application as data corruption. The verification library could detect the mismatch between amount of data received and expected.

We have created a test suite for the verification library. This test suite contains some typical errors, like setting the send and receive counts wrong for an `MPI_Alltoallv` operation:

```
for (i = 0; i < nbr_processes; i++) {
  send_count[i] = i;
  recv_count[i] = nbr_processes - i; /* ERROR: should be 'my_rank' */
}
```

Here, each process should receive an amount of data proportional to its rank. However, this requires that each process sets all entries of the `recv_count` array to its own rank.

Test cases for MPI-IO deal with using process-individual file views with shared file pointers (which is not allowed by the MPI standard), or defining non-contiguous file views where the extents of gaps are not multiples of the extent of the elementary data type. The latter problem is local to each process and only occurs within a collective operation.

The MPI/SX verification library detects all these errors. It is more interesting to observe how MPI libraries without an explicit verification capability handle such problems. We checked this with

our own MPI library and with the latest version of MPICH2 (1.0.2), which can be considered as a reference implementation. Some problems lead to deadlocks with both libraries, while some, like the `MPI_Alltoallv` problem described above, behave differently.

This problem returned an (unspecified) `MPI_ERR_TRUNCATE` error code with our library, but deadlocked with MPICH2. The reason for this deadlock was found in the chosen algorithm: In MPICH2, empty messages are not sent within `MPI_Alltoallv`, However, in this case, the receiver erroneously waits for a (non-empty) message. The algorithm in our library is different and thus reacts differently. With the verification library, the following diagnostic message is output:

```
VERIFY MPI_ALLTOALLV(0): sendsize[1]=4 != expected recvsize(1)[0]=16
VERIFY MPI_ALLTOALLV(1): sendsize[0]=0 != expected recvsize(0)[1]=12
VERIFY MPI_ALLTOALLV(2): sendsize[0]=0 != expected recvsize(0)[2]=8
VERIFY MPI_ALLTOALLV(3): sendsize[0]=0 != expected recvsize(0)[3]=4
```

Some of the verifications done within the MPI-IO operations are local. As the overhead is actually not very large and some of these functions like `MPI_File_set_view` are not as performance critical as the collective communication functions, we decided to activate most verifications also in the default version of the MPI/SX library. Still, the extended diagnostic messages are only printed by the verification library. The standard library only returns an error code which needs to be decoded by `MPI_Error_string`. However, this leads to the effect that 8 of the 9 problems for MPI-IO are also detected without the verification library when using MPI/SX. The original ROMIO as used in MPICH2, too, generates an error for one of the problems tested.

6. Verification overhead

Naturally, the verification functionality incurs extra overhead. The nominal overhead per collective operation, measured as number of additional collective operations, is from two to eight operations. The amount of data exchanged per operation is small, in the worst case proportional to the number of processes in the communicator, and in most cases just a single `MPI_Aint`.

6.1. Synthetic Benchmark

We used a standard benchmark for collective communication operations to measure the actual overhead of the verification library on different platforms. Although the verification does not only take place in the communication operations, the methods used and thus the overhead implied is very similar.

We express this overhead as a *relative slowdown*, which means we give a percentage of how much additional time is needed for a collective operation to complete at all processes. Figure 1 shows these numbers for runs of this benchmark on an NEC SX-8 machine, using 32 processes on 4 nodes and for an IA-32 based cluster with Myrinet 2000 interconnect, using 32 processes on 16 nodes.

As can be seen from the charts, the overhead for small message sizes, which means short execution times of the non-verified operation, is very significant. It varies between a factor of 4 for `MPI_Allgather` up to a factor of 14 for `MPI_Gather` and `MPI_Scatter`. The high overhead for the latter operations relates to the relatively short execution time of the non-verified operation.

With increasing data size, the execution time of the non-verified operation increases, while the time required for the verification remains constant. This leads to a decrease of the relative overhead of the verification library. Summarizing, the overhead of the verification library is non-negligible for individual collective operations with small data.

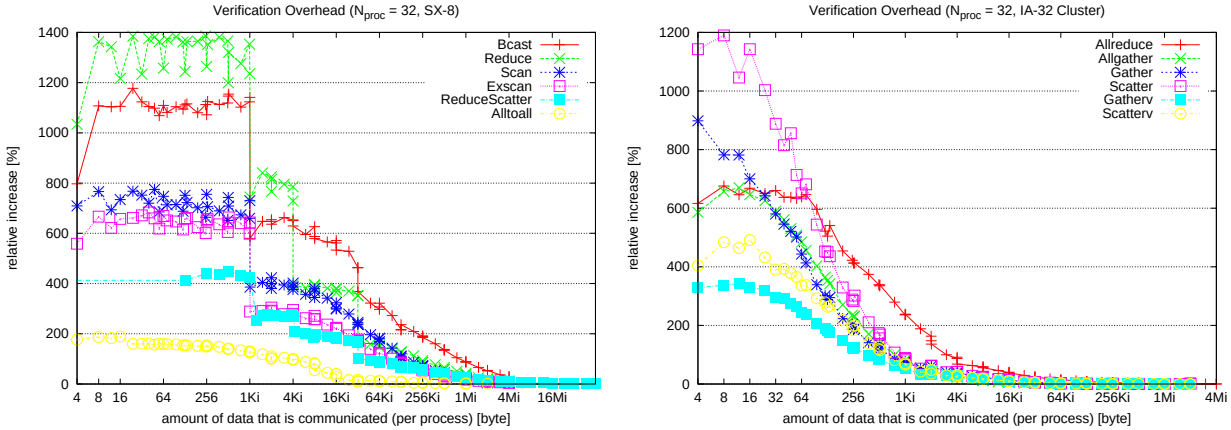


Figure 1. Relative overhead of verification library for some collective operations on NEC SX-8 and generic IA-32 cluster

6.2. Application Benchmark

The overhead of the verification library as evaluated in the previous section is significant, especially for small data sizes and collective operations with a short native run time. It is obvious that for applications where the performance is bound by the performance of the MPI collectives for small data, use of the verification library imposes a significant overhead. We wanted to know how much this actually affects the run time of an actual application.

For this evaluation, we chose the *hypre* [2] package of preconditioners and linear solvers. We ran the test case `ij` from the test suite using the default algebraic multi-grid solver (index 0) with a $50 \cdot N_{proc} \times 100 \times 100$ grid, with N_{proc} being the number of processes. In this configuration, each process has a peak memory usage of about 1GB. The only difference between the two executables was the additional option `-lvmpi` when linking the application.

We executed the test case on 8 nodes of the aforementioned cluster, with each node running 2 processes ($N_{proc} = 16$). To get statistically valid data, we performed 99 runs of each variant of the test case. In each `ij` run, more than 5000 collective operations are executed. About 70% of these operations are `MPI_Allreduce` with 4-byte integer values. The remaining operations are calls to `MPI_Allgather(v)` and `MPI_Gather` with small data sizes below 128 bytes. Based on the results of the synthetic benchmarks, we estimated the overhead that would be introduced by using the verification library to about 2s additional runtime. The runtime with the standard MPI library is in the range of 140s. The results of our tests are presented in Table 2. We give the average (arithmetic mean) of all run times and the related standard deviation as well as the 90^{th}-quantile to better handle the outliers.

The overhead calculated from the average execution time is very small, and it contrasts to the expected overhead of about 2 seconds. The reason for this may be found in the value of the standard deviation which is larger than the expected overhead. Calculating the overhead based on the 90^{th}-quantile results in a value closer to the expected overhead.

The large standard deviation surprised us as the cluster was operated as a production system with a batch system in which the nodes are exclusive for the individual application. The distribution plot in Figure 2 shows that the execution times actually vary between 137 and 164 seconds for the same

Table 2
Run time of *hypre* test case `ij` with and without verification library.

	w/o verify	w/ verify	overhead
avg. runtime [s]	139.78	140.10	0.32
standard deviation	2.96	3.86	n/a
90^{th}-quantile [s]	141.64	142.27	0.63

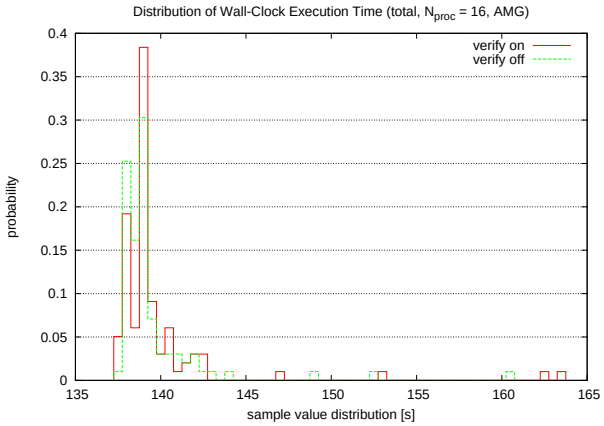


Figure 2. Normalized distribution of execution time of *hypre* `ij` test case.

problem on the same set of nodes. This shows two things: first, simply averaging across a small number of runs can easily give bogus results, and secondly that in this case, the actual overhead of the verification library could thus even be tolerated for a production environment.

7. Conclusion

The MPI/SX verification library for collective operations is an easy-to-use tool to verify the correctness of an MPI application. It covers all MPI functions which need to be called by all processes of an inter- or intra-communicator to complete. Because our implementation can access internal data structures of the MPI/SX library, it is able to cover more potential errors than a portable approach reasonably could.

We continue to work on the verification library in order to create a comprehensive tool not only for collective operations, but for the complete range of MPI functions. To achieve this, we will implement techniques to ensure the correct use of MPI datatypes, and will extend the verification to include non-collective MPI operations. To validate the correct use of MPI datatypes, we will make use of the available technique in MPI/SX for the space- and time-efficient representation of derived datatypes [10]. In contrast to approaches that use a hash-value of a datatype, using the complete representation of a datatype can never lead to false diagnostics or undetected errors. Aspects of verification for non-collective operations are correct use of message buffers and MPI requests. Another important aspect, namely the detection of deadlocks, is already implemented in MPI/SX outside the scope of the verification library.

8. Acknowledgments

We thank our colleague Jens Georg Schmidt for support with the application benchmarking. All data for the performance evaluation was processed using *perfbase* [13].

References

[1] Jayant DeSouza, Bob Kuhn, and Bronis R. de Supinski. Automated, scalable debugging of MPI programs with Intel Message Checker. In *Second International Workshop on Software Engineering for High Performance Computing System Applications in conjunction with 27th International Conference on Software Engineering*, 2005.

[2] R.D. Falgout and U.M. Yang. hypre: a library of high performance preconditioners. In *Computational Science (3) - ICCS 2002*, volume 2331 of *Lecture Notes in Computer Science*, pages 632–641, 2002.

[3] Chris Falzone, Anthony Chan, Ewing Lusk, and William Gropp. Collective error detection for MPI collective operations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 12th European PVM/MPI Users' Group Meeting*, volume 3666 of *Lecture Notes in Computer Science*, pages 138–147, 2005.

[4] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. *MPI – The Complete Reference*, volume 2, The MPI Extensions. MIT Press, 1998.

[5] Bettina Krammer, Katrin Bidmon, Matthias S. Müller, and Michael M. Resch. MARMOT: An MPI analysis and checking tool. In *Parallel Computing (ParCo)*, 2003.

[6] Bettina Krammer, Matthias S. Müller, and Michael M. Resch. MPI application development using the analysis tool MARMOT. In *International Conference on Computational Science (ICCS)*, volume 3038 of *Lecture Notes in Computer Science*, pages 464–471, 2004.

[7] Bettina Krammer, Matthias S Müller, and Michael M. Resch. MPI I/O analysis and error detection with MARMOT. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 11th European PVM/MPI Users' Group Meeting*, volume 3241 of *Lecture Notes in Computer Science*, pages 242–250, 2004.

[8] Glenn R. Luecke, Hua Chen, James Coyle, Jim Hoekstra, Marina Kraeva, and Yan Zou. MPI-CHECK: a tool for checking fortran 90 MPI programs. *Concurrency and Computation: Practice and Experience*, 15(2):93–100, 2003.

[9] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI – The Complete Reference*, volume 1, The MPI Core. MIT Press, second edition, 1998.

[10] Jesper Larsson Träff, Rolf Hempel, Hubert Ritzdorf, and Falk Zimmermann. Flattening on the fly: efficient handling of MPI derived datatypes. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 6th European PVM/MPI Users' Group Meeting*, volume 1697 of *Lecture Notes in Computer Science*, pages 109–116, 1999.

[11] Jesper Larsson Träff and Joachim Worringen. Verifying collective MPI calls. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 11th European PVM/MPI Users' Group Meeting*, volume 3241 of *Lecture Notes in Computer Science*, pages 18–27, 2004.

[12] Jeffrey S. Vetter and Bronis R. de Supinski. Dynamic software testing of MPI applications with Umpire. In *Supercomputing (SC)*, 2000. `http://www.sc2000.org/proceedings/techpapr/index.htm#01`.

[13] Joachim Worringen. Experiment management and analysis with perfbase. In *Proceedings of the IEEE International Conference on Cluster Computing*, Boston, September 2005. IEEE Computer Society Press.

Capturing Petascale Application Characteristics with the Sequoia Toolkit

J. S. Vetter[a] *, N. Bhatia[a], E. M. Grobelny[b], P. C. Roth[a]

[a]Future Technologies Group, Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN 37831 USA

[b]High-Performance Computing and Simulation Research Laboratory, Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL 32611 USA

Characterization of the computation, communication, memory, and I/O demands of current scientific applications is crucial for identifying which technologies will enable petascale scientific computing. In this paper, we present the Sequoia Toolkit for characterizing HPC applications. The Sequoia Toolkit consists of the Sequoia trace capture library and the Sequoia Event Analysis Library, or SEAL, that facilitates the development of tools for analyzing Sequoia event traces. Using the Sequoia Toolkit, we have characterized the behavior of application runs with up to 2048 application processes. To illustrate the use of the Sequoia Toolkit, we present a preliminary characterization of LAMMPS, a molecular dynamics application of great interest to the computational biology community.

1. Introduction

The Future Technologies Group [1] at Oak Ridge National Laboratory performs basic research in core technologies for future generations of high-end computing systems. An important aspect of our work involves the characterization of existing scientific applications to understand their demands for computation, communication, memory, and I/O. By understanding the demands of current applications, we hope to gain insight regarding which technologies will best satisfy the needs of those applications in the future.

Building on experience with the mpiP profiling library [2], we have developed the Sequoia Toolkit to support characterization of applications that use the Message Passing Interface [3] for communication. The Sequoia Toolkit includes an event tracing facility and scripts to process and manage event trace files. The Sequoia tracing facility captures performance data for computation and communication activity in an application. Sequoia records events for each MPI call made by the application, including data such as the number of bytes transferred. For computation, Sequoia uses the Performance Application Programming Interface [4] (PAPI) to collect hardware counter metrics describing the computation that occurs between successive calls to MPI functions.

Using scripts provided with the Sequoia Toolkit, event trace files can be analyzed to compute ratios for characterizing application performance, such as the ratio of the number of floating point operations to the amount of data transferred between MPI tasks. Sequoia trace files can also be used as input to architectural simulators; the Sequoia Toolkit includes a simple simulator that models the system interconnect using the LogP [5] model. We envision using Sequoia event trace files as input for performance modeling approaches such as Modeling Assertions [6]. Because the process

*This research was sponsored by the Office of Mathematical, Information, and Computational Sciences, Office of Science, U.S. Department of Energy under Contract No. DE-AC05-00OR22725 with UT-Battelle, LLC. The Petascale Execution Time Evaluation project is supported by the office of Science of the U.S. Department of Energy. Accordingly, the U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

of using trace files for simulation is restricted to the processor count of the original trace file parameters, we are actively developing a communication extrapolation tool that can generate synthetic communication traces directly from empirical trace files, which were captured at smaller processor counts.

To facilitate development of tools that use Sequoia event trace files, we are developing the Sequoia Event Analysis Library, or SEAL. SEAL is a C++ library that provides a tool-independent infrastructure for opening Sequoia event trace files, decoding their contents, and dispatching control to tool-specific functions that process events (e.g., to find the total number of bytes transferred during an application run).

In the next section, we describe the Sequoia event tracing library and SEAL. In Section 3, we discuss some of our current characterization work on MPI applications of interest to Oak Ridge National Laboratory and the United States Department of Energy. We discuss related, existing tracing facilities and analysis tools in Section 4, and summarize our work on the Sequoia Toolkit in Section 5.

2. The Sequoia Toolkit

The Sequoia Toolkit includes an event tracing library, scripts for management and basic processing of Sequoia event trace files, and SEAL, the Sequoia Event Analysis Library. In this section, we describe these Sequoia Toolkit components.

2.1. The Sequoia Event Tracing Library

The Sequoia event tracing library captures the behavior of a running application by writing *event records* to event trace files. Sequoia event trace files are ASCII text files containing a chronological sequence of event records describing computation and communication events that occurred during the run of a target application. Although Sequoia includes scripts for basic trace file management and processing, Sequoia trace files are ASCII text files so that common text-processing utilities, like `sed` and `awk`, can be used to manipulate them. Sequoia generates one trace file per MPI process, but toolkit scripts can be used to merge these files into a composite event trace file. Metadata describing the format of event trace records is kept in a separate file called `events.stdef`. A portion of the `events.stdef` is shown in Listing 1. This file can be modified to enable the processing of new event types or to extend existing event types. That figure shows the file's header, several definitions of built-in MPI types, communicators, and operations, and definitions for a few of the functions in the MPI API.

The Sequoia tracing library operates on the MPI profiling interface. Using this interface, the Sequoia library intercepts calls to each MPI function and records information such as the sender or receiver of the operation and the number of bytes transferred. Because the MPI profiling interface is defined as part of the MPI-1 and MPI-2 standards, Sequoia can be used with any compliant MPI implementation. Also, due to the interface's design, no source code modifications are required to use Sequoia to trace a MPI application. The application does need to be re-linked against the Sequoia library, and small source code modifications may be used to control the event trace volume and to support fine-grained characterization of the computation between communication events.

Sequoia event records can be broadly classified into three categories:

- *MPI event records*—records that describe entry to or exit from a MPI function;
- *Computation event records*—records that describe computation between successive MPI function calls; and
- *Special event records*—records that describe tracing library meta events.

Listing 1: A portion of a Sequoia `events.stdef` file.

```
@ SEQUOIA
@ Command :
@ Version                  : 0.9
@ Build date               : Jul 19 2005, 17:10:29
@ ST env var               : [null]
@ Final Trace Dir          : .
@ MPI Type                 : MPI_BYTE 2 1
@ MPI Type                 : MPI_CHAR 4 1
@ MPI Type                 : MPI_DOUBLE 14 8
...
@ MPI Comm                 : MPI_COMM_WORLD 0 1
@ MPI Comm                 : MPI_COMM_SELF 1 1
@ MPI Comm                 : MPI_COMM_NULL -1 0
@ MPI Op                   : MPI_MAX 0
@ MPI Op                   : MPI_MIN 1
...
@ EDEF ts rank thread pcid Init X hostname size starttime
@ EDEF ts rank thread pcid Finalize E
@ EDEF ts rank thread pcid Comp E
@ EDEF ts rank thread pcid Comp X cycles insts loads stores flops fmas ipc
@ EDEF ts rank thread pcid Send E  count datatype dest tag comm
@ EDEF ts rank thread pcid Send X  count datatype dest tag comm
...
```

Table 1
Fields common to all Sequoia trace file event records.

Name	Explanation
ts	Event timestamp
rank	MPI rank in which event occurred
thread	ID of the thread in which event occurred (*reserved*)
pcid	Program counter id (*reserved*)
type	Type of event
mode	Whether record specifies a transition *into* a state ('`E`') or *out of* a state ('`X`')

These three categories are described in more detail in the remainder of this section. Regardless of an event record's category, certain fields are present at the start of each record. These fields are described in Table 1. The values of the `type` and `mode` fields determine the type and number of the remaining fields in the event record. In effect, these two fields allow Sequoia-based tools to look up the correct event record definition from the `events.stdef` file. The `thread id` and `pcid` fields are reserved for future use.

2.1.1. MPI Event Records

Sequoia MPI event records indicate an entry into or exit from a MPI library routine. Each MPI event record contains data relevant to the specific MPI function that was called. For instance, the event record for the `MPI_Init` call includes fields for the name of the host running the MPI process, the total number of MPI processes in the run, and the start time of the application. The record for

Table 2
Computation event record fields, depending on the value of the `ST` environment variable.

Value	Fields
c 0	cycles instructions loads stores fp-ops fp-mas ipc
c 1	cycles instructions loads stores *reserved reserved* ipc
c 2	cycles instructions *reserved reserved* fp-ops fp-mas ipc

Table 3
Explanation of computation event record fields.

Name	Explanation
cycles	Number of CPU cycles in the computation interval
instructions	Number of instructions executed
loads	Number of load instructions executed
stores	Number of store instructions executed
fp-ops	Number of floating point operations performed
fp-mas	Number of floating point multiply-add instructions performed
ipc	Instructions executed per cycle

the `MPI_Send` call includes fields for the number of items sent, the type of the data sent (using an integer encoding specified in the `events.stdef` file), the receiving process' MPI rank, and the message tag and MPI communicator used for the operation.

2.1.2. Computation Event Records

Sequoia event records for computation capture hardware performance counter data for the computation that occurred in a MPI process between successive communication events. By default, Sequoia generates one pair of computation event records that describe the computation between successive calls to routines in the MPI interface. With small modifications to the application source code as described in Section 2.1.3, hardware counter data can be recorded with finer granularity. A computation entry event record is generated when a process exits any MPI routine. When the process makes its next MPI routine call, Sequoia reads the hardware counters and generates a computation exit record that captures the values.

For portability, Sequoia uses the PAPI library for obtaining the hardware counter data. Sequoia also resets the hardware counters after sampling, so that the values in computation event records indicate differences between samples instead of cumulative values. Sequoia uses the `ST` environment variable to specify which hardware counters it will sample for computation event records. If `ST` is not set, "`c 0`" is the default value. Table 2 shows the event record fields used for the three recognized `ST` values; Table 3 explains the possible event record fields.

2.1.3. Special Event Records

Sequoia can be used to trace MPI applications without modifying their source code, however, in this mode, Sequoia may generate trace files of unmanageable size, especially when tracing long-running applications that transition frequently between computation and communication operations. However, given the regular structure of many scientific applications, insight into the behavior of scientific applications can often be obtained by observing a small number of iterations of an application's main loop (e.g., a small number of simulation time steps in a climate simulation code).

To enable the user to control the trace event record volume, Sequoia tracing can be disabled and then re-enabled as an application process executes. Sequoia event tracing is disabled when the `MPI_Pcontrol` function is called with an argument of 0, and re-enabled when `MPI_Pcontrol` is called with an argument of 1. The action of disabling and re-enabling Sequoia tracing is recorded with a special event trace record without parameters.

By default, Sequoia generates one pair of computation event records for all computation between successive MPI calls in a process. However, the computation that occurs in that interval may not be uniform. To allow users to trace application computation at finer granularity, Sequoia can insert `Mark` event records into the event trace. Inserting a `Mark` event record into the event trace terminates the current computation interval (hence sampling the hardware performance counters as described in Section 2.1.2) and starts a new computation interval. To generate `Mark` event records, the user inserts `MPI_Pcontrol` calls into the application source code with an argument greater than 1. This argument value is recorded as part of the `Mark` event record and can be considered to be a computation phase identifier by analysis tools.

2.2. The Sequoia Event Analysis Library

Tools that consume Sequoia event trace files as input can have many purposes, but they each share some similar operations: they read Sequoia trace file records, decode the records, and dispatch control to other parts of the code that will process the records. Because this functionality is not specific to any one tool, it can be factored into a library that is shared by many tools. We are developing the Sequoia Event Analysis Library, or SEAL, that provides a tool-independent infrastructure for reading Sequoia event trace files. SEAL provides a C++ API and implements a tool framework for opening Sequoia event trace files, reading event records in order from the trace file, and dispatching control to tool-specific functions that operate on the current event record. A SEAL-based tool need only provide the tool-specific event record processing functions.

3. Application Characterization

The Sequoia Toolkit is under active development, but we have already used it to trace and characterize several scientific applications of interest to Oak Ridge National Laboratory and the U.S. Department of Energy, including:

- *AMBER*—a suite of programs that allow users to perform molecular dynamics operations, particularly on bio-molecules.
- *GYRO*—a simulation of microturbulance in tokamak fusion reactors.
- *HYCOM*—an ocean general circulation model.
- *LAMMPS*—a classical molecular dynamics simulation code.
- *POP*—a 4D ocean modeling code.
- *SPPM*—a 3D gas dynamics simulation code.
- *SWEEP3D*—a neutron transport code.
- *UMT2K*—a 3D photon transport code.

For example, we have used the Sequoia Toolkit to characterize the computation and communication demands of the LAMMPS molecular dynamics code. The Sequoia trace files were gathered during a run on the IBM p690+ at Forschungszentrum Jülich in Germany. Sequoia tracing was enabled for two simulation time steps. Using simple post-processing scripts to accumulate performance data from the Sequoia event trace file, we generated the application characterization shown

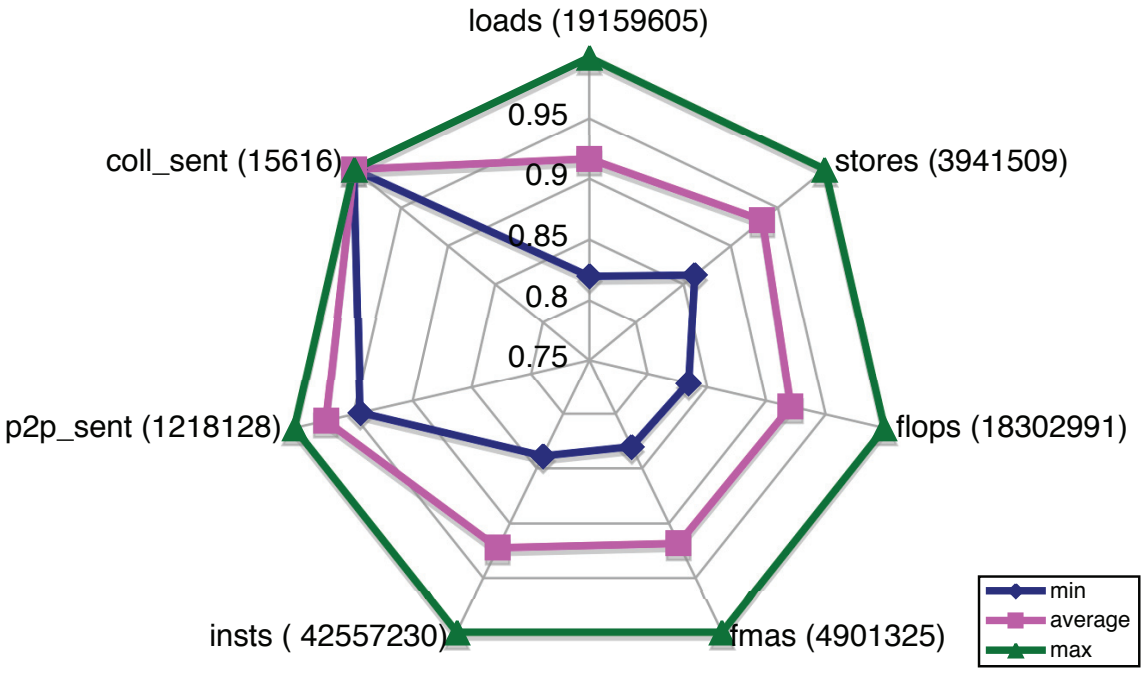


Figure 1. Characterization of LAMMPS for 64 tasks using data collected using the Sequoia Toolkit *Values are normalized using the maximum value observed for any task during the run. Axis labels include the maximum value.*

in Figures 1–3. For each performance metric in Figure 1, we plot the minimum, maximum, and average value for each computation or communication interval represented in the event trace file (i.e., for each pair of MPI event records or computation event records in the event trace file for each application process). To improve readability of the chart, all values were normalized to the interval [0,1]. We characterize LAMMPS using several common performance ratios in Figures 2 and 3. In Figure 2, we compare LAMMPS running with 16, 128, and 512 processes for one user-defined phase of the program's run. The axes show the number of point-to-point communication operations (p2p), bytes transferred per floating point operation for point-to-point communication (p2bpf), number of collective communication operations (coll), bytes transferred per floating point operation for collective operations (collbpf), loads and stores per floating point operation (lspf), and instructions per cycle (ipc). For readability, the values are normalized using performance data across all user-defined phases. Figure 3 shows the same data but normalized using only values from the phase being analyzed. The characterization shows that LAMMPS places significantly different demands on a computing system at the three process counts we considered.

4. Related Work

Event tracing is a well-established approach for collecting performance data that describes an application's behavior, and trace analysis tools are often developed in conjunction with a trace capture library. Examples of current performance analysis tools that adopt this approach include Vampir and Vampirtrace [7], the KOJAK [8] automated performance analysis tool with its EPILOG tracing facility, and the Tuning and Analysis Utilities [9] (TAU) that has its own tracing facility but also can operate on Vampir event trace files (or can convert them to a format it can recognize). We are

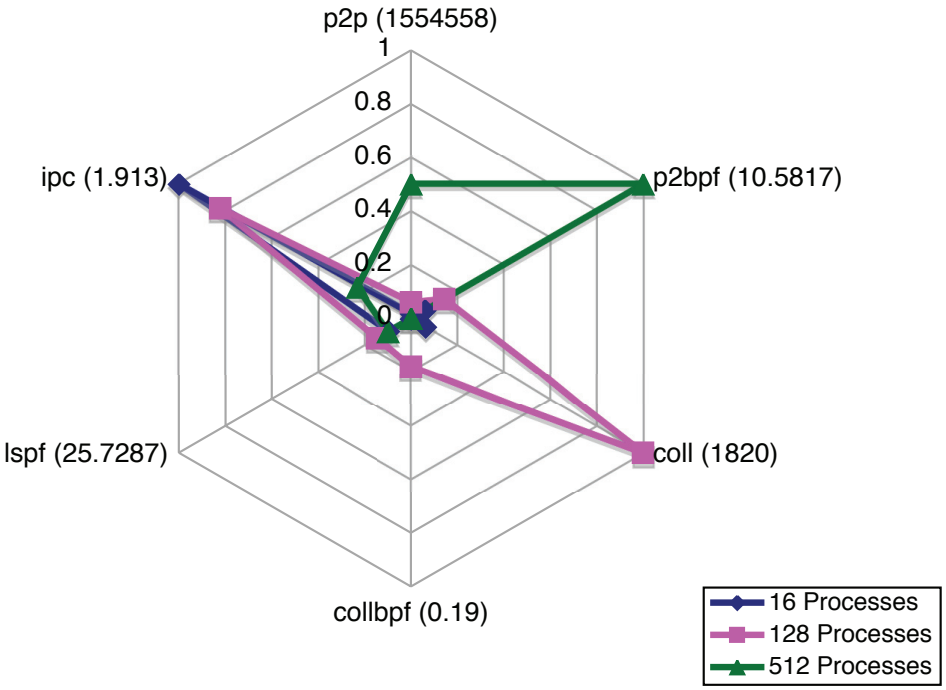


Figure 2. Characterization of LAMMPS phase 5
Metric values are normalized using values from all phases.

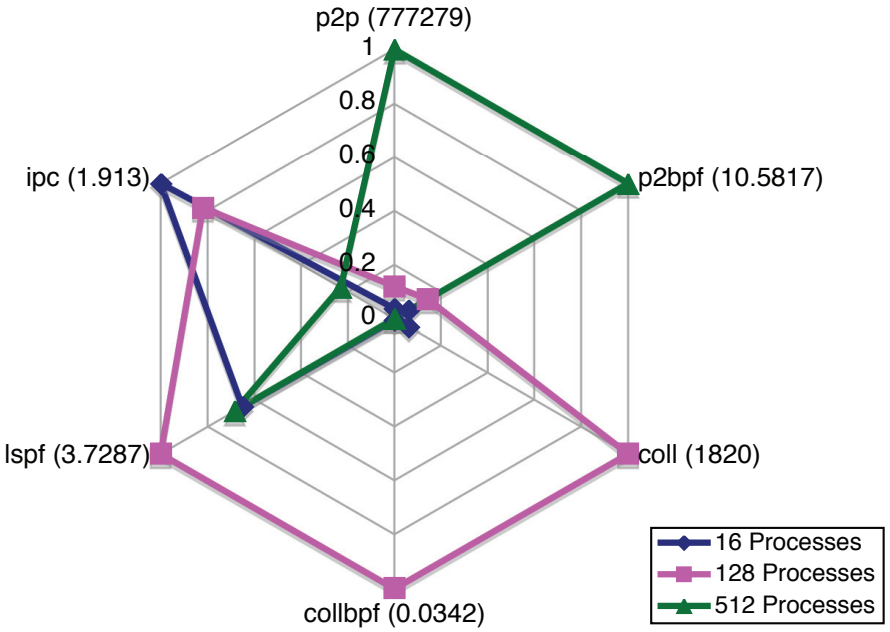


Figure 3. Characterization of LAMMPS phase 5, normalized within that phase.
This figure shows the same data as Figure 2, but the values are normalized using the phase 5 values only.

actively working on the integration of the Sequoia Toolkit with some of these tools. For example, we are currently working on a converter between the Sequoia event trace file format and EPILOG, to allow analysis of Sequoia event traces by the KOJAK tool set.

5. Summary

As a starting point for understanding the demands of future applications on future computing architectures, the Future Technologies Group at Oak Ridge National Laboratory seeks to understand the behavior of current scientific applications. To support this effort, we are developing the Sequoia Toolkit. Sequoia includes an event tracing library that records data about both MPI communication events and the computation that occurs between those communication events. Sequoia also includes scripts for management and basic analysis of Sequoia event trace files. Recently, we have started work on the Sequoia Event Analysis Library, a C++ library that facilitates the development of tools for analyzing Sequoia event traces. We have traced and analyzed scientific applications from a variety of domains, including computational biology, materials science, nuclear fusion, and climate modeling.

We continue to refine and extend the Sequoia Toolkit to make application characterization easier for the end user, to support new analyses, and to improve integration with existing analysis tools. For instance, we are modifying Sequoia to sample hardware counter data for communication entry events in addition to communication exit events, to support analysis of the internal computational requirements of MPI routines. Also, we are actively working on a utility that converts Sequoia event trace files to formats that enable analysis and visualization using the popular TAU and Vampir tools.

References

[1] ORNL Future Technologies Group. http://www.csm.ornl.gov/ft/, October 2005.

[2] J. S. Vetter and M. O. McCracken. Statistical scalability analysis of communication operations in distributed applications. In *PPoPP '01: Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 123–132, New York, NY, USA, 2001. ACM Press.

[3] W. Gropp, R. Thakur, and E. Lusk. *Using MPI-2: Advanced Features of the Message Passing Interface*. MIT Press, Cambridge, MA, USA, 1999.

[4] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, Washington, DC, USA, November 2000. IEEE Computer Society.

[5] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, E. E. Santos, K. E. Schauser, R. Subramonian, and T. von Eicken. LogP: A practical model of parallel computation. *Communications of the ACM*, 39(11):78–85, 1996.

[6] S. R. Alam and J. S. Vetter. Multiresolution performance modeling with modeling assertions, in submission 2005.

[7] W.E. Nagel, A. Arnold, M. Weber, H.C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, 1996.

[8] B. Mohr and F. Wolf. KOJAK - a tool set for automatic performance analysis of parallel programs. In *Euro-Par*, volume 2790 of *Lecture Notes in Computer Science*, pages 1301–1304, Heidelberg, 2003. Springer-Verlag.

[9] S. Shende, A. D. Malony, J. Cuny, P. Beckman, S. Karmesin, and K. Lindlan. Portable profiling and tracing for parallel, scientific applications. In *SPDT '98: Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 134–145, New York, NY, USA, 1998. ACM Press.

Author Index

AUTHOR INDEX

Aguilar, X. 869
Aldana, J. 735
Aldegunde, M. 439
Aldinucci, M. 73, 89, 145
Aliaga, J. 333
Almeida, F. 333, 375
Alonso, J.M. 647
an Mey, D. 431
Andersson, U. 179
Andrade, D. 565
André, F. 65, 89
Angiolini, F. 745
Arafeh, B. 41
Argollo, E. 465
Arnal, J. 245
Arnau, V. 623
Arnold, G. 447
Asenjo, R. 557
Asenov, A. 407
Atienza, D. 549, 753, 769
Bacigalupo, D. 163
Badía, J.M. 333
Bandera, G. 541
Barabas, L. 211
Barba, J. 777
Barrachina, S. 333
Bartic, A. 761
Beaudoin, A. 391
Bencsura, A. 121
Benini, L. 745
Benoit, A. 423, 673, 787
Berthold, J. 835
Bertozzi, D. 745
Bhatia, N. 917
Birkeland, O.R. 691
Bisseling, R.H. 819
Blanco, V. 333
Blochinger, W. 301
Boldarev, A.S. 351, 359
Bond, W. 583
Buenabad-Chávez, J. 507
Buisson, J. 65, 89
Cai, X. 383
Campa, S. 89
Canot, E. 399
Cantero, M.C. 591
Carmueja, M.G. 565
Carretero, J. 523
Carta, S. 745
Castillo, M. 333
Catalyurek, U. 3
Catthoor, F. 549, 573
Chateau, T. 663
Chen, W. 631
Chetverushkin, B.N. 343, 351, 359
Chick, J.P. 423
Claudio, D. 317
Clematis, A. 195, 607, 655
Clément, F. 811
Coghlan, B.A. 129
Cole, M. 673, 787
Coppola, M. 89, 531
Corana, A. 195
Corbalán, J. 81
Corbera, F. 557
Corporaal, H. 761
Cuenca, J. 229
Currle-Linde, N. 49
D`Agostino, D. 607, 655
Danalis, A. 481
Dandamudi, S. 499
Danelutto, M. 73, 89,145, 681, 803
Day, K. 41
Dazzi, P. 681
De Micheli, G. 753
De Dreuzy, J.R. 391
de Dieuleveult, C. 399
de Sande, F. 333, 375
de-Andrés, E. 735
del Castillo, N. 261
Delgado, C. 375
Deng, X. 631
DeSouza, J. 901
Di Cosmo, R. 811
Dietze, H. 717
Diverio, T. 317
Domínguez-Domínguez,S. 507
Dominiczak, S. 367
Drummond, T. 325
Duarte, A. 465, 473
Dünnweber, J. 787
D'yachenko, S.V. 359
Erhel, J. 391, 399
Falcou, J. 663
Fernández, J.J. 727
Fernández, P. 269
Fishgold, L. 481
Flesch, I. 819
Fraguela, B.B. 565
Frings, W. 367
Fürlinger, K. 15
Galiano, V. 325
Galizia, A. 607

Gallopoulos, E. 309
García, C. 599, 615
García, F. 523
García, I. 269, 727
García, L.-P. 229
García, R. 375
García Quiñones, C. 27
García-Loureiro, A.J. 407, 439
Garzón , E.M. 727
Gasilov, V.A. 351, 359
Geisler, S. 105
Genko, N. 753
Gerndt, M. 15
Giaccherini, G. 73
Gianuzzi, V. 655
Gibbon, P. 367
Gilmore, S. 673
Giménez, D. 229
Giménez, J. 869
Glimsdal, S. 383
Gómez, J.A. 293
Gómez, S. 261
González, A. 27
González, P. 869
Gorlatch, S. 787
Granado, J.M. 293
Grelck, C. 859
Grobelny, E.M. 917
Gross, S. 431
Guim, F. 81
Hartmann, O. 457
Hastings, S. 3
Haszpra, L. 121
He, L. 163
Heine, F. 113
Hermanns, M.-A. 885
Hermida, R. 769
Hernández, V. 221, 647
Herruzo, E. 541
Hidalgo-Herrero, M. 843
Higgins, R. 171
Hillston, J. 673
Hölbig, C. 317
Horányi, A. 121
Hovestadt, M. 113
Huang, K. 3
Iakobovski, M.V. 351
Imasaki, K. 499
Iwashita, T. 237
Jacobs, J. 583
Jarvis, S. 163
Jorba, J. 155
Joubert, G.R. 105
Juckeland, G. 877
Jurie, F. 663
Kacsuk, P. 121
Kalna, K. 407
Kao, O. 113, 491
Kartashova, E.L. 351, 359
Keller, A. 113
Kereku, E. 15
Kollias, G. 309
Konovalov, A. 901
Krammer, B. 893
Krawczyk, H. 699
Krukov, V. 901
Kuchen, H. 795
Kuhn, B. 901
Kühnal, A. 885
Kühnemann, M. 457
Kullmann, L. 121
Kumar, V.S. 3
Kurc, T. 3
Küster, U. 49
Labarta, J. 81, 869
Lagzi, I. 121
Langella, S. 3
Langtangen, H.P. 383
Lastovetsky, A. 171
Lendvay, Gy. 121
León-Hernández, C. 285
Li, E. 631
Lippert, T. 447
Llort, G. 869
Loogen, R. 835
Lopes, P.A. 515
López, J.C. 777
Lovas, R. 121
Luque, E. 155, 465, 473, 623
Madajczak, T. 699
Madriles, C. 27
Magán, I. 769
Malony, A. 203
Marcuello, P. 27
Marescaux, T. 761
Margalef, T. 155
Marín, I. 623
Martin, V. 811
Martínez, E. 869
Martínez, P. 591
Mayo, R. 333
Medeiros, P.D. 515
Meloni, P. 745
Mendías, J.M. 769
Mészáros, R. 121
Mifune, T. 237

Migallón, H. 245
Migallón, V. 245, 325
Minkin, A.S. 351
Miranda-Valladares, G. 285
Mohr, B. 187, 367, 885
Moltó, G. 647
Morandi Jr., P. 317
Morris, A. 203
Moya, F. 777
Mucci, P. 179
Müller, M.S. 893
Müller-Pfefferkorn, R. 211
Mustapha, H. 391
Nagel, W.E. 211, 717, 877
Narayanan, S. 3
Navarro, A. 557
Navas-Delgado, I. 735
Nedland, M. 691
Neumann, R. 211
Nollet, V. 761
Nudd, G. 163
Nussbaum, D. 277
Obata, N. 237
Olcoz, K. 549
Olkhovskaya, O.G. 359
Ortega-Mallén, Y. 843
Ortigosa, P.M. 269
Oster, S. 3
Ouyang, Z. 631
Pan, T. 3
Pardines, I. 253
Pascual-Montano, A. 615
Paternesi, A. 145
Patvarczki, J. 121
Pazat, J.-L. 65
Pedersen, G.K. 383
Pelegrín, B. 269
Pelych, D. 163
Penadés, J. 245, 325
Peña, R. 851
Peón, M. 769
Pérez, A.J. 735
Pérez, R.M. 591
Perez Ramas, J.B. 769
Pesciullesi, P. 531
Pflüger, S. 877
Piñuel, L. 573, 599
Plata, O. 541
Plaza, A. 591
Plaza, J. 591
Poldner, M. 795
Pollock, L. 481
Polyakov, S.V. 351
Pombo, J.J. 439
Pomplun, N. 447
Pouls, R. 583
Presti, L. 531
Prieto, M. 573, 599, 615
Quintana, E.S. 333
Quintana, G. 333
Raffo, L. 745
Ramírez , S. 735
Ranaldo, N. 137
Rångevall, A. 761
Rauber, T. 457
Ravazzolo, R. 145, 531
Redondo, J.L. 269
Reichelt, V. 431
Rerrer, U. 491
Resch, M.M. 49, 893
Rexachs, D. 465, 473
Richter, M. 447
Rincón, F. 777
Ripoll, A. 623
Rips, S. 97
Risio, B. 49
Rivera, F.F. 253
Rodero, I. 81
Rodrigues de Souza, J. 465
Rodríguez, C. 333
Román, J.E. 221
Romero, L.F. 415
Roth, P.C. 917
Rubio, F. 843
Rünger, G. 457
Rutt, B. 3
Ryan, J.P. 129
Sack, J.-R. 277
Saltz, J.H. 3, 639
Samofalov, V. 901
Sanchez, J.E.P. 709
Sánchez, J. 27
Sánchez, J.M. 293
Santos, A. 333
Scholz, S.-B. 859
Schöne, R. 877
Segura, C.M. 851
Seoane, N. 407
Sérot, J. 663
Servat, H. 869
Shende, S.S. 203
Shilnikov, E.V. 343
Shimasaki, M. 237
Singh, D.E. 253, 523
Smit, G.J.M. 583
Snøve Jr., O. 691

Spiegel, A. 431
Spooner, D. 163
Streit, A. 57
Swany, M. 481
Szyld, D. 309
Tabik, S. 727
Tenllado, C. 573, 599
Terboven, C. 431
Tineo, A. 557
Tiskin, A. 827
Tomás, A. 221
Torquati, M. 73
Touzene, A. 41
Träff, J.L. 909
Trelles, M.A. 415
Trelles, O. 415, 735
Trenkler, B. 717
Tretola, G. 137
Trystram, D. 709
Turányi, T. 121
Ujaldón, M. 639
Valencia, D. 591
Valko, V.V. 359
Vanneschi, M. 73, 145, 531
Vega, M.A. 293
Velasco, J.M. 549
Vetter, J.S. 917
Villa, D. 777
Villanueva, F.J. 777
Vodicka, A. 811
Wäldrich, O. 57
Weis, P. 811
Wieder, P. 57
Wloch, R. 877
Wolf, F. 187, 885
Worringen, J. 909
Wylie, B.J.N. 187
Yang, X.Y. 623
Ye, H. 277
Zapata, E.L. 541, 557
Zhang, X. 3
Zhang, Y. 631
Zheltov, S. 901
Ziegler, W. 57
Zimeo, E. 137
Zoccolo, C. 89

Already published:

Modern Methods and Algorithms of Quantum Chemistry - Proceedings
Johannes Grotendorst (Editor)
Winter School, 21 - 25 February 2000, Forschungszentrum Jülich
NIC Series Volume 1
ISBN 3-00-005618-1, February 2000, 562 pages
out of print

Modern Methods and Algorithms of Quantum Chemistry - Poster Presentations
Johannes Grotendorst (Editor)
Winter School, 21 - 25 February 2000, Forschungszentrum Jülich
NIC Series Volume 2
ISBN 3-00-005746-3, February 2000, 77 pages
out of print

Modern Methods and Algorithms of Quantum Chemistry - Proceedings, Second Edition
Johannes Grotendorst (Editor)
Winter School, 21 - 25 February 2000, Forschungszentrum Jülich
NIC Series Volume 3
ISBN 3-00-005834-6, December 2000, 638 pages
out of print

Nichtlineare Analyse raum-zeitlicher Aspekte der hirnelektrischen Aktivität von Epilepsiepatienten
Jochen Arnold
NIC Series Volume 4
ISBN 3-00-006221-1, September 2000, 120 pages

Elektron-Elektron-Wechselwirkung in Halbleitern: Von hochkorrelierten kohärenten Anfangszuständen zu inkohärentem Transport
Reinhold Lövenich
NIC Series Volume 5
ISBN 3-00-006329-3, August 2000, 146 pages

Erkennung von Nichtlinearitäten und wechselseitigen Abhängigkeiten in Zeitreihen
Andreas Schmitz
NIC Series Volume 6
ISBN 3-00-007871-1, May 2001, 142 pages

Multiparadigm Programming with Object-Oriented Languages - Proceedings
Kei Davis, Yannis Smaragdakis, Jörg Striegnitz (Editors)
Workshop MPOOL, 18 May 2001, Budapest
NIC Series Volume 7
ISBN 3-00-007968-8, June 2001, 160 pages

Europhysics Conference on Computational Physics - Book of Abstracts
Friedel Hossfeld, Kurt Binder (Editors)
Conference, 5 - 8 September 2001, Aachen
NIC Series Volume 8
ISBN 3-00-008236-0, September 2001, 500 pages

NIC Symposium 2001 - Proceedings
Horst Rollnik, Dietrich Wolf (Editors)
Symposium, 5 - 6 December 2001, Forschungszentrum Jülich
NIC Series Volume 9
ISBN 3-00-009055-X, May 2002, 514 pages

Quantum Simulations of Complex Many-Body Systems: From Theory to Algorithms - Lecture Notes
Johannes Grotendorst, Dominik Marx, Alejandro Muramatsu (Editors)
Winter School, 25 February - 1 March 2002, Rolduc Conference Centre, Kerkrade, The Netherlands
NIC Series Volume 10
ISBN 3-00-009057-6, February 2002, 548 pages

Quantum Simulations of Complex Many-Body Systems: From Theory to Algorithms- Poster Presentations
Johannes Grotendorst, Dominik Marx, Alejandro Muramatsu (Editors)
Winter School, 25 February - 1 March 2002, Rolduc Conference Centre, Kerkrade, The Netherlands
NIC Series Volume 11
ISBN 3-00-009058-4, February 2002, 194 pages

Strongly Disordered Quantum Spin Systems in Low Dimensions: Numerical Study of Spin Chains, Spin Ladders and Two-Dimensional Systems
Yu-cheng Lin
NIC Series Volume 12
ISBN 3-00-009056-8, May 2002, 146 pages

Multiparadigm Programming with Object-Oriented Languages - Proceedings
Jörg Striegnitz, Kei Davis, Yannis Smaragdakis (Editors)
Workshop MPOOL 2002, 11 June 2002, Malaga
NIC Series Volume 13
ISBN 3-00-009099-1, June 2002, 132 pages

Quantum Simulations of Complex Many-Body Systems: From Theory to Algorithms - Audio-Visual Lecture Notes
Johannes Grotendorst, Dominik Marx, Alejandro Muramatsu (Editors)
Winter School, 25 February - 1 March 2002, Rolduc Conference Centre, Kerkrade, The Netherlands
NIC Series Volume 14
ISBN 3-00-010000-8, November 2002, DVD

Numerical Methods for Limit and Shakedown Analysis
Manfred Staat, Michael Heitzer (Eds.)
NIC Series Volume 15
ISBN 3-00-010001-6, February 2003, 306 pages

Design and Evaluation of a Bandwidth Broker that Provides Network Quality of Service for Grid Applications
Volker Sander
NIC Series Volume 16
ISBN 3-00-010002-4, February 2003, 208 pages

Automatic Performance Analysis on Parallel Computers with SMP Nodes
Felix Wolf
NIC Series Volume 17
ISBN 3-00-010003-2, February 2003, 168 pages

Haptisches Rendern zum Einpassen von hochaufgelösten Molekülstrukturdaten in niedrigaufgelöste Elektronenmikroskopie-Dichteverteilungen
Stefan Birmanns
NIC Series Volume 18
ISBN 3-00-010004-0, September 2003, 178 pages

Auswirkungen der Virtualisierung auf den IT-Betrieb
Wolfgang Gürich (Editor)
GI Conference, 4 - 5 November 2003, Forschungszentrum Jülich
NIC Series Volume 19
ISBN 3-00-009100-9, October 2003, 126 pages

NIC Symposium 2004
Dietrich Wolf, Gernot Münster, Manfred Kremer (Editors)
Symposium, 17 - 18 February 2004, Forschungszentrum Jülich
NIC Series Volume 20
ISBN 3-00-012372-5, February 2004, 482 pages

Measuring Synchronization in Model Systems and Electroencephalographic Time Series from Epilepsy Patients
Thomas Kreutz
NIC Series Volume 21
ISBN 3-00-012373-3, February 2004, 138 pages

Computational Soft Matter: From Synthetic Polymers to Proteins - Poster Abstracts
Norbert Attig, Kurt Binder, Helmut Grubmüller, Kurt Kremer (Editors)
Winter School, 29 February - 6 March 2004, Gustav-Stresemann-Institut Bonn
NIC Series Volume 22
ISBN 3-00-012374-1, February 2004, 120 pages

Computational Soft Matter: From Synthetic Polymers to Proteins - Lecture Notes
Norbert Attig, Kurt Binder, Helmut Grubmüller, Kurt Kremer (Editors)
Winter School, 29 February - 6 March 2004, Gustav-Stresemann-Institut Bonn
NIC Series Volume 23
ISBN 3-00-012641-4, February 2004, 440 pages

Synchronization and Interdependence Measures and their Applications to the Electroencephalogram of Epilepsy Patients and Clustering of Data
Alexander Kraskov
NIC Series Volume 24
ISBN 3-00-013619-3, May 2004, 106 pages

High Performance Computing in Chemistry
Johannes Grotendorst (Editor)
Report of the Joint Research Project:
High Performance Computing in Chemistry - HPC-Chem
NIC Series Volume 25
ISBN 3-00-013618-5, December 2004, 160 pages

Zerlegung von Signalen in unabhängige Komponenten: Ein informationstheoretischer Zugang
Harald Stögbauer
NIC Series Volume 26
ISBN 3-00-013620-7, April 2005, 110 pages

Multiparadigm Programming 2003
Joint Proceedings of the
3rd International Workshop on Multiparadigm Programming with Object-Oriented Languages (MPOOL'03)
and the
1st International Workshop on Declarative Programming in the Context of Object-Oriented Languages (PD-COOL'03)
Jörg Striegnitz, Kei Davis (Editors)
NIC Series Volume 27
ISBN 3-00-016005-1, July 2005, 300 pages

Integration von Programmiersprachen durch strukturelle Typanalyse und partielle Auswertung
Jörg Striegnitz
NIC Series Volume 28
ISBN 3-00-016006-X, May 2005, 306 pages

OpenMolGRID - Open Computing Grid for Molecular Science and Engineering
Final Report
Mathilde Romberg (Editor)
NIC Series Volume 29
ISBN 3-00-016007-8, July 2005, 86 pages

Computational Nanoscience: Do It Yourself!
Lecture Notes
Johannes Grotendorst, Stefan Blügel, Dominik Marx (Editors)
Winter School, 14. - 22 February 2006, Forschungszentrum Jülich
NIC Series Volume 31
ISBN 3-00-017350-1, February 2006, 528 pages

NIC Symposium 2006 - Proceedings
G. Münster, D. Wolf, M. Kremer (Editors)
Symposium, 1 - 2 March 2006, Forschungszentrum Jülich
NIC Series Volume 32
ISBN 3-00-017351-X, February 2006, 384 pages

All volumes are available online at

http:// www.fz-juelich.de/nic-series/.